AWKを256倍使うための本

志村 拓・鷲北 賢・西村克信　共著

アスキー出版局

AWKを256倍使うための本

志村拓・鷲北賢・西村克信　共著

アスキー出版局

商　標

MS-DOSは、米国 Microsoft 社の登録商標です。
UNIXは、UNIX System Laboratories, Inc.が開発しライセンスしています。
jgawkのもとになるgawkの著作権は、Free Software Foundation, Inc.が保持しています。
その他、本書に掲載されている会社名、製品名は、一般に各メーカーの登録商標または商標です。
なお本文中には、TM、®は明記していません。

表紙作成　荒井桂子
本文イラスト　小林直子

はじめに

ほんとうに、コンピュータを使って仕事の効率をあげたいのなら、たとえどんな些細な仕事でも、

しっかりコンピュータにやらせるべき

である。コンピュータとは、こちらが思っているよりはるかにわがままで、自分のできる仕事しかやろうとせず、不得手な仕事をすべて人間に押しつけてくる。自分では気づかないうちに、逆にコンピュータに使われていたりするのだ。データベースの吐いたデータのなかから、エディタでシコシコ不必要なデータを消したりしているなんてその典型である。

超ダサダサじゃないか！

最近、こうした仕事をコンピュータ自体にちゃんと行わせるために注目されているのが、ミニ言語やマクロ言語といった比較的言語仕様の小さなプログラミング言語だ。これらの言語は、インタープリタまたはそれに準ずるもので、コンパイルなしで状況に応じたプログラムを速攻で書くことができる。コンピュータの投げ出した仕事をすぐさまコンピュータに突き返してやることができるのだ。

そんなミニ言語のなかでも、速攻性、柔軟性、機能性のすべてにおいて

最右翼に位置する逸品がawkである。

本書は、スクリプトの書き方から、文法、プログラマーズTIPS、サンプルプログラム、リファレンスまでawkのすべてを網羅した一冊である。その内容は、

256倍シリーズと名乗るに恥ずかしくない

濃一い内容となっている。awk の初心者はもちろん、awk のスクリプトはひととおり書けると勘違いしている中級者も、本書によって awk の真の姿に一歩近づけるはずだ。本書を読破して awk をマスターし、コンピュータの仕事の好き嫌いをねじ伏せることができれば、あなたのコンピュータライフは確実に 1 ランクアップするだろう。

1993 年 7 月　筆者を代表して　　志村　拓

目 次

はじめに 3
目 次 5

awk はここまで使え!

awk のある生活 11
もうアプリなんていらない! 27
正規表現うわー! 42

•awk の変遷について 49
•awk の入手法について 50

awk を解剖する!

awk の背骨 53
awk スクリプトの骨格 53
awk の基本動作 55
パターンの役割 56
BEGIN, END パターン 59

アクションの役割 62
awk の歯顎 64
レコードとフィールド 64
awk の胃袋たち 71
awk の変数 71
"連想配列" は awk 独特の機能 72
連想配列の実体 77
ユーザー定義関数 79
変数のスコープ管理 80

awk の舞台裏を探る!

—— プログラマーズ TIPS ——

文字列を自在に 85
変数を自在に 94
awk をより速く 109
検索を自由に 116
レコードとフィールド 123
ファイルを自在に 126
こんなことも… 137

awk 実践プログラミング!

—— 100 行野郎の実際 ——

awk 住所録 143
ここだよーんゲーム 160
姓名バリバリチューン!! 174

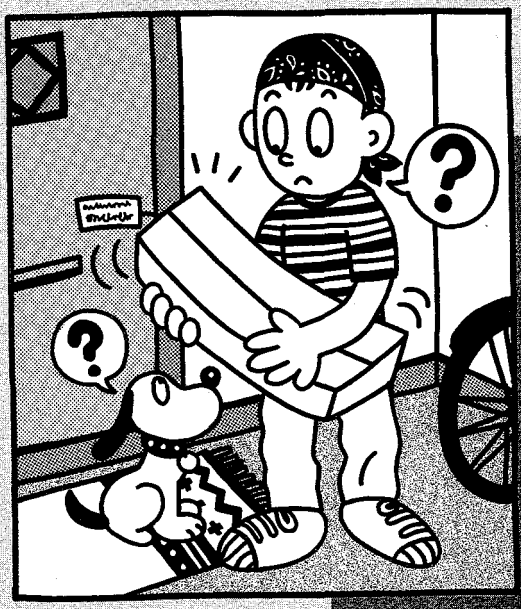
ユーザー管理 awk *184*
ジャパン・ダイレクトだ Jpn2awk!! *201*
立体視 awk だ!! *209*
awk 表プロセッサ *215*

APPENDIX

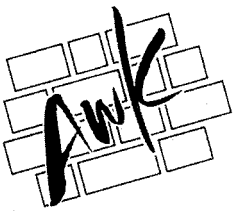
文法のまとめ *225*
リファレンス *230*

著者紹介 *277*
奥付 *278*

awkはここまで使え!



「awkって、よく聞くケド、いったい何なの？」といった向きのために、本章はawkの入門編的説明を中心に話を進めていく。初心者の読者はとにかく内容が解らなくとも、とりあえず本章を読み切ってみてほしい。また、初心者でない向きのために、各所にウンチク話を仕込んでおいたので、まったくの紙の無駄にはなっていないはずだ。さて、それではそろそろ始めることにしよう。



awkのある生活

なぜawkが必要なの?

「コンピュータを使うと、コンピュータに行わせたい仕事が増える」

なにやらマーフィーの法則のようなこの文句は、筆者が最近実感していることの1つである。たとえば、いくつかのテキストファイルをどこかに持っていくとき、そのファイルの合計がフロッピーディスクに入りきるか計算して調べなければならない。こんな計算はコンピュータを使っていなければ不要である。物理的な従来のメディアであれば「ちょっと分量がかさ張るから、縮小コピーして持っていこう」で済んでしまうはずだ。コンピュータの世界とは、

結局数値ドリブンな環境なのである。

コンピュータを使いながら脇に電卓をおいて計算しているといった風景は、こうした環境を如実に物語っている。

これらは見方を変えると、現在のパソコンがユーザーの望んでいる仕事をこなしてくれていないということにほかならない。SF映画の1シーンにでてくるように［今月私が何時間働いたか報告しろ］と言うと、［あなたは実質5時間しか働いていません］といった具合であれば、特に［コンピュータにさせたい仕事］が増えたといった感覚はないのだ。要するに現代のコンピュータが現在吐き出すデータは、まだまだ不完全なものということだが、かといって

計算機の報告した値を電卓で計算している

なんてあまりにも情けない。まるで［オレは、ここまでだから後はヨロシク］とコンピュータに使われているようだ。ほんとにダサダサだ。こうしたちょっとした仕事もコンピュータにやらせるべきだ。そうでなければ、

筋ってぇものがとおらない。

そこでテキストデータの処理なら抜群の能力を持つ“awk”の登場となるわけだ。

コンピュータ利用状況を知る

「あなたは今月、何時間コンピュータを使いましたか?」

こんな質問をされたら、あなたは答えることができるだろうか？

パソコンの場合、コンピュータを使って課金されるものといってもせいぜい電気代くらいのものだ。これが汎用機のような時間貸しのシステムともなると、はるかにシビアとなる。CPU の使用時間は秒単位で記録されるし、ディスクの上に置いている（といっても物理的な意味ではない）ファイルの量に関しても記録がとられ、秒いくら、キロバイトいくらといった具合にお金をとられるのだ。

こうした課金に脅えずに、好きなだけコンピュータをブン回せるのがパソコンの利点だからあまり細かいことは気にしたくないが、自分がどれくらいコンピュータを使っているかぐらいは把握しておくと何かと便利だ。たとえば、「今月は先月よりコンピュータの使用時間が長いから電気代はいくらかかる」とか、「この原稿を書くのに、コンピュータを使って○○時間もかかってしまった」とか、いろいろと使えるわけだ。

そこで、理由は聞かずにだまされたと思って**リスト 1–1**に示すファイルを、MS–DOS を起動するディスクのルートディレクトリに“sec1970.awk”という名前で作ってみてほしい。

リスト 1–1　<sec1970.awk>

```
BEGIN{ srand(); printf "%10d ", srand() }
```

これはawkのスクリプトなのだが、間違いなくできたか調べるために図1–1のようにawkを実行してみよう。以下のように9桁の数字がでてきたらたぶんうまくできている。もし、awkが何かエラーメッセージを出した場合はスクリプトファイル名を間違えたかスクリプトファイルの中身が間違っている可能性があるので、もう一度確認してみよう。「コマンドまたはファイル名が違います。」と表示された場合は、残念ながら本書の説明の領域を外れてしまっている。この本を読む前にawkを入手するか、MS–DOSのPATHコマンドの使い方を調べることをお勧めする。

```
A>jgawk -f ¥sec1970.awk
 736419712
A>
```

図 1–1　<sec1970.awk>を実行

参考までに説明しておくと、この数値は1970年1月1日から現在までの経過時間を秒で表したものだ。大阪万博の年から現在までの経過時間が秒でカウントしてもたかだか十進数で9桁そこそこで表されてしまう。「人間の一生なんて地球の歴史に比べたら微々たるもの」などという天文学者の言葉が妙に説得力のあるものに思えてしまう（さらに図1–1の数字を解析することによって、筆者がこの原稿をどんな境遇の下で書いているかも理解できるハズだ。シクシクシク）。

AUTOEXEC.BATで開始時間を記録

さて話を先に進めよう。このスクリプトを使ってコンピュータの利用時間を記録する。コンピュータの利用開始時間は、autoexec.batに組み込むことでいいだろう。図1–2のようにして、リダイレクションを使用して“compacct.log”というファイルに、awkの出力を貯め込むようにしておく。これでコンピュータに火を入れれば、コンピュータの利用開始時間が書き込まれるハズだ。jgawkの前にあるechoコマンドの行はちょっとした“おまじない”なので同じように書き込んでおくこと。

```
@ECHO OFF

REM Setting for MS-DOS ver.5.0
REM --------------------------
Set PathBase=A:¥usr¥ucb¥pds;A:¥usr¥ucb¥ascii;A:¥usr¥pds;A:¥bin¥
bat;A:¥bin¥dos;A:¥usr¥bin
REM Path %PathBase%;A:¥bin¥win;A:¥var¥bcc¥bin;A:¥var¥tex¥bin;A:
¥var¥AWKcc
Path %PathBase%;A:¥Var¥just¥JUST5;A:¥Var¥Just¥Taro5;A:¥VAR¥JUST
¥JW2
Set PathAppl=
Set TEMP=H:¥
Set DOSDIR=A:¥BIN¥DOS
Set DIRCMD=/OGN /L

echo . >> A:¥compacct.log
jgawk -f A:¥sec1970.awk >> A:¥compacct.log

                                    :
                                    :
```

図1–2　AUTOEXEC.BATの一例

さて、コンピュータを使い終わった時の時間であるが、残念ながらMS–DOSにはコンピュータの電源を切る時に自動的に実行される命令は存在しない。そこでshutdown.batなるバッチファイルを用意して、必ずコンピュータの電源を消す前に実行するようにする（もちろん、これを忘れず

に行うのはあなた自身だ)。

このshutdown.batにハードディスクのヘッドをリトラクトするようなコマンドや、CD-ROMを排出するような命令、またハードディスクのキャッシュをハードディスクに書き込む命令などを一緒に書いておくとマシンの安全のためにもよいだろう。筆者が使用しているshutdown.batを**リスト1-2**に示す。

リスト1-2　筆者の<shutdown.bat>

```
@echo off

jgawk -f A:¥sec1970.awk >> A:¥compacct.log

cdej cd_101
smartdrv /C
shipping
```

これで準備はOKだ。あとはコンピュータの電源を入れたり切ったりするにつれて(切る時はshutdownコマンドを忘れないように)、その時の時間がどんどん貯まっていくはずだし、compattr.logは利用開始時間、終了時間がスペースで区切られて一行になるはずだ。めでたしめでたし... あれ? これ何の本だっけ?

利用時間を求める

というわけで、ここからが本題なのだ。できあがったcompacct.logファイルを集計して、実際に現在までに何時間コンピュータを使ったか集計してみることにしよう。そう、こんな処理を行うのにawkはもってこいのツールなのだ。まずはcompacct.logから利用時間を集計するスクリプト“acct1.awk”をご覧いただくことにしよう。

リスト 1–3　利用時間を集計するスクリプト <acct1.awk>

```
#
# /compacct.logから現在までに何秒コンピュータを利用したか集計する。

NF == 3 {
    total = total + $2 - $1
}

END {
    print "現在までのコンピュータ利用時間は、" total "秒です。"
}
```

試しに acct1.awk を実行して、compacct.log の集計をとってみることにしよう。awk は**リスト 1–4**のようにしてスクリプトファイルとデータファイルを指定する。これによって awk はスクリプトファイルに書かれたスクリプトを解釈しながらデータを処理するのだ。データファイルは省略することも可能で、省略した場合は処理するデータを標準入力から読み込む。データファイルとして指定できるのはテキストファイルだけで制御コードのゴッチャリ入ったバイナリファイルは処理できない。これは awk の仕様だ。

リスト 1–4　スクリプトファイルとデータファイルの指定

```
A>jgawk -f <スクリプトファイル> [データファイル ...]
```

フツー、スクリプトで".awk"でしょ。

話はそれるが、awk のプログラムはプログラムと呼ばずに、「スクリプト」と呼ぶ。プログラムとスクリプトはどちらも同じ意味であるが、原則的にプログラムに比べスクリプトは小さな言語仕様のプログラム言語の場合に使用されるようだ。たとえば UNIX のコマンドインタープリタである csh や sh のプログラムはシェルスクリプトと呼ぶし、sed や perl もスクリ

プトと呼ぶのが普通だ。そういう意味ではMS-DOSのバッチはバッチスクリプトと呼びそうだが、そうは言われずバッチプログラムと呼ばれる。まぁ実際にはどちらを慣用的に使用しているかによって決まるんだけど…。

「慣用的に」と言えば、このリストのファイル名は“smpl1.awk”となっており、拡張子として“.awk”が使用されているが、これは特にこだわる必要はない。C言語やアセンブラのように拡張子を省略した場合に“.C”や“.ASM”が使用されるといった機能は、awkのスクリプトには特に用意されていない。awkのスクリプトファイルの拡張子を“.awk”にするのは単に慣習的に、「フツー“.awk”でしょ」といった程度のことだ。

COMMAND.COMのお陰?

話が脱線しているついでに説明しておくと、awkではリスト1-4に示したようにスクリプトをファイルとして用意して実行する方法の他に**リスト1-5**のように直接コマンドラインにスクリプトを書く方法も用意されている。この方法は、単純で短いスクリプトの場合に有効で、いちいちファイルを作らなくてすむので便利だ。たとえばリスト1-1のようなスクリプトは、ズバリこの方法を使う局面なのだ。が、本書ではこの方法は使用しない。

リスト1-5　awkスクリプトをawkに実行させる

```
A>jgawk <スクリプト> [データファイル ...]
```

これは、MS-DOSのコマンドインタプリタであるcommand.comのコマンド解釈機能がチープだからだ。awkのスクリプトでは、リダイレクションやパイプで使用する“<”、“>”や“|”の記号をスクリプト内で使用するが、これをMS-DOSのコマンドラインから渡そうとする場合、気をつけないとMS-DOSのコマンドとして解釈されてしまい、

非常に危険だ。

shやcshでは、“’”で括った中ではこれらのメタキャラクタも単なる文字列であり、パラメータとして渡されるので通常UNIXでawkのスクリプトをコマンドパラメータで記述する場合は“’”で括って書くが、MS-DOSにはこのような機能が“””しかないのだ。たとえば以下のような簡単なスクリプトでさえ、MS-DOSの環境ではコマンドラインからスクリプトを指定するのは難しいのだ。

```
A>jgawk "BEGIN{print "<<hello>>¥n" > hello}"
文法が違います.

A>jgawk 'BEGIN{print "<<hello>>¥n" > hello}'
jgawk.exe:  syntax error near line 2:
(END OF FILE)
^ parse error

A>
```

図1-3　MS-DOSのcommand.comって...

まだこれだけなら何とかなるかも知れないが、これにMS-DOSのコマンドラインをマルチステートメントに拡張するようなTSRを組み入れていて、“;”をコマンドセパレータとして使用しているような場合はさらに話が複雑だ。というのも、awkのコマンドセパレータも“;”であり、この文字も先のメタキャラクタと同じようにawkにパラメータとして渡るように配慮しなければならないからだ。

ダッサいCOMMAND.COM!!

どうしてもMS-DOSでコマンドラインからawkのスクリプトを指定したいという向きは、ダサいCOMMAND.COMを捨てて（といっても、config.sysのshellとして指定できるものは少ないので“del command.com”するわけにもいかないだろうが）、GNUのbashやその他のフリーウェアのshモドキを使用することをお勧めする。

幸先の悪いスタート

話をもとに戻すことにしよう。acct1.awk の実行結果が図 1–4だ。

```
A>type a:¥compacct.log

 736439620  736439717 .
 736439763  736440070 .
 736440114  736441728 .
 736447715  736454101 .
 736478343 .
 736481806 .
 736487422  736502152 .
 736508112 .
 736519929  736519987 .
 736525063
A>jgawk -f acct1.awk a:¥compacct.log
jgawk.exe: warning: record not terminated
 input line number 10, file 'a:¥compacct.log'
 source line number 5, file 'acct1.awk'
現在までのコンピュータ利用時間は、23192秒です。

A>
```

図 1–4　<acct1.awk>で集計してみる

何やら awk がブーたれているようだが、とりあえずこれを無視すれば、なんとかまっとうに動いているようだ。なにやら幸先の悪いスタートを切ってしまったが、まぁこんなもんだろう。

awk のブーイングが気になるが、一応スクリプトが動いたので acct1.awk の説明をしておくことにしよう。

データをブッタ斬れ!

awkはデータファイルから1行分のデータを$0という変数に読み込む。これをawkではレコードと呼ぶ。次にこれをスペースで区切られたフィールドに分解する。分解された各フィールドの文字列は、左側から順に$1,$2,$3...という変数に代入される（本当はこれらの処理は動的に行われるのだが、ここでは話を簡単にするためにこのように説明する）。これらの変数は通常のプログラミング言語よろしくアクセスすることができる。また、この時に変数NFにフィールドの数が代入される。

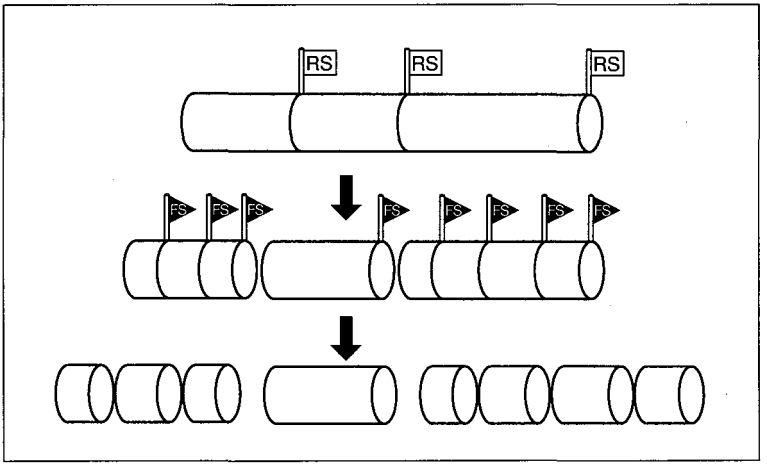


図1-5　レコードとフィールド

“{”と“}”で囲む命令

awkの命令は、必ず“{”と“}”で囲む。これはawkスクリプトのキマリだから「なぜ？」と聞かれても困ってしまうが、ここは特別に、お客さんにだけその理由をお教えすることにしよう。

お客さんだけね

さて、この問いに答える前に、awkがどのようにスクリプトを実行しているか説明しなければならない（なんだか説明のネストが深くなってるなぁ）。スクリプトに書かれた命令は、データのレコード数回繰り返し実行される。簡単に言えば、awkはデータファイルから1レコード（つまり1行のデータ）を読み込んでは、そのデータをもってスクリプトを頭から実行し、スクリプトの終わりまで達したら次のレコードを読み込んで、またスクリプトの頭から実行していくのだ。

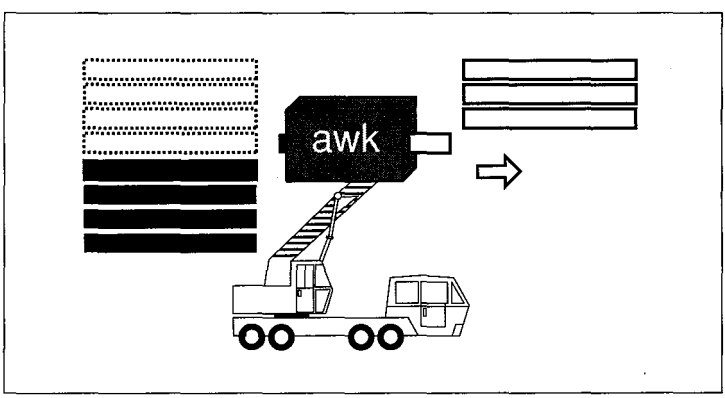


図1-6　パターンとアクション

データファイルから1レコードのデータを取得してスクリプトをawkが実行する際にスクリプトを実行するか否かを条件として与えられる。この条件はawkの命令を囲む“ {”の前に書く。たとえば、**リスト1-6**は、フィールドの数が5個の場合に“ {”と“ }”で囲まれた命令が実行され、そのレコードが出力される。リスト1-6の“NF ＝＝ 3”や“END”がこの条件にあたるものだ。“ {”と“ }”で命令を囲むのは、こうした条件を満たした場合、どこからどこまでの命令を実行するか識別するためにあるわけだ。ちなみに条件は省略することも可能で、その場合そのなかの命令はすべてのレコードで実行される。

acct1.awkの動作を調べる

awkの基礎講習が終わったので、ようやくacct1.awkの説明ができる。と思ったら、acct1.awkのリストは、すでに数ページ彼方ではないですか。「おーい、説明しやすいようにもう一度acct1.awkのリストを出しとくれー」あぁ一人芝居は情けない…。

リスト1-6　<acct1.awk>

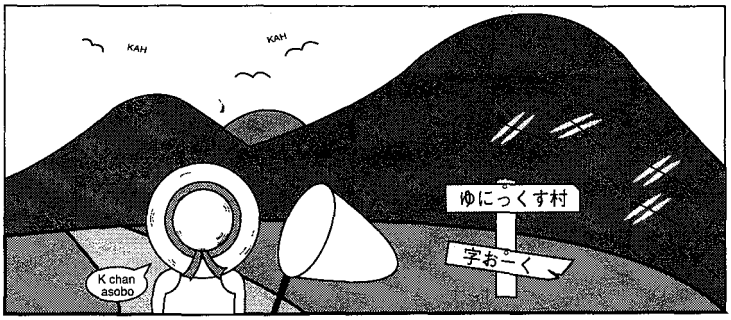
```
#
# /compacct.logから現在までに何秒コンピュータを利用したか集計する。

NF == 3 {
        total = total + $2 - $1
}

END {
        print "現在までのコンピュータ利用時間は、" total "秒です。"
}
```

まず、1行目と2行目は注釈でプログラム的には何の意味もない。awkでは“#”から行末までは注釈として無視されるのだ。

またも話はそれるが世の中の言語、特にawkのようなスクリプトやデータファイルにはコメントマークとして“#”を使用するものが少なくない。これは偶然の一致ではない。その理由はawkの故郷、UNIXにある。

awkの故郷‥‥UNIX

UNIXのコマンドインタプリタであるBourneシェルやCシェル（newcshやkshやtcshもみーんな）も“#”をコメントマークにしている。また、/etc/fstab（おっと、最近ではvfstabだが）、/usr/lib/crontab、/etc/hosts、Makefile等のシステムの使用するデータファイルも“#”コメントマークだ。



つまり、UNIXのシステムでは、

“#”は非常に典型的なコメントマーク

なのである（最近では、.Xdefaultsや Imakefile 等のように、Cコンパイラのプリプロセッサに前処理をさせるものは、“#”以外をコメントマークにしているものもあるが）。筆者は、awkのコメントマークが“#”になったのもこれらの仕様に合わせたのだろうと推測している。

さらにBSD系のUNIXには、

実行属性のついたファイルを実行する際にそのファイルがCOFF（UNIXの実行ファイルのフォーマットの呼び名。最近のSVR4系UNIXでは、COFFフォーマットに変わってELFフォーマットが使用されている。）であるか否かを調べ、もしCOFFフォーマットを表すマジックナンバーが見つからなかった場合、シェルスクリプトとして認識、実行される。この時、1行目に“#! <実行ファイル名>”を書いておくと、Bourneシェルを起動せずに<実行ファイル名>を実行し、標準入力からそのスクリプトに書かれている内容が入力されるといった仕様になっている。このあたりもコメントマークが“#”だと非常に都合がよい。ちなみにMS-DOS上でもこれに似た機能を実現するための“#!”というコマンドがフリーウェアで公開されているので、興味のある向きは一度試してみるといいだろう。

あーまた脱線してしまった!

解説に戻ろう。4行目から6行目までの命令は、4行目に書いてある条件が成り立つ場合に実行される。具体的に言えば条件式“NF == 3”のNFはフィールドの数を表すシステム変数なので、フィールドの数が3個の場合に5行目の“total = total + $2 - $1”が実行されることになる。“total = total + $2 - $1”のtotalは変数、$2は2番目のフィールド、$1は1番目のフィールドの内容を表すシステム変数だ。つまり、2番目のフィールドの内容から1番目のフィールドを引いた値を変数totalに加算しているわけだ。

ここで、再度compacct.logの内容を見てみることにしよう。awkはこのファイルを1行読み込んでは、それをスペースで区切られたフィールドに分解してスクリプトを実行する。これは先に説明したとおりだが、compacct.logの場合1番目のフィールドには利用開始時間が、2番目には利用終了時間が入ることになるわけだ。利用終了時間から利用開始時間を引けば、利用時間が得られる。この結果をどんどんtotalという変数に加算しているのが5行目の“total = total + $2 - $1”なわけだ。

```
利用開始時間   利用終了時間
     ↓            ↓

  736439620   736439717 .
  736439763   736440070 .
  736440114   736441728 .
  736447715   736454101 .
  736478343 .                    shutdown コマンドを忘れたので、
  736481806 .                    終了時間が記載されていない
  736487422   736502152 .
  736508112 .
  736519929   736519987 .
  736525063                      現在の利用開始時間。shutdown コマンド
                                 を実行すると、この数値の右に終了時間
                                 が記入される
```

図 1-8　<compacct.log>の内容

図1–8を見てもわかるとおり、shutdownコマンドの実行を忘れるとその時の利用終了時間が記録されない。このような行では2番目のフィールドは“．”ということになってしまい利用時間を計算できない。ということで4行目に“NF ＝＝ 3 {”と書いてフィールドが3つある場合しか利用時間の計算を行わないようにしているわけだ。図1–2AUTOEXEC.BATのjgawkの前に書いたおまじない、“echo．>>A:¥compacct.log”のわけがこれでわかってもらえるだろう。

8行目の“END”は、8行目の“{”と10行目の“}”で囲まれた命令（簡単に言えば9行目に書かれた命令のことなんだけどネ）を、データファイルをすべて読み終えたあとで実行するという意味だ。9行目は、変数totalの値をメッセージとともに画面に表示している。つまりファイルをすべて読み終えた時点で変数totalには利用時間の合計が入っているのでそれを表示しようということだ。

「秒単位では、何がなんだかわからない!」

という向きは、9行目を以下のようにすれば時間単位の表示も可能だ。

```
print "現在までのコンピュータ利用時間は、" total/3600 "時間です。"
```

ちなみに“END”の正反対の動きをするものとして“BEGIN”という条件がある。これは、データファイルを読む前にそれ以降の命令を実行するというものだ。変数の初期化を行うような場合に便利な条件だ。

awkのブーイングの理由

ひととおりacct1.awkの説明をしたので、最後になぜacct1.awkを実行した際にawkがぶーたれたかを説明しておくことにしよう。

ぶーたれた理由は・・・

実は、これはスクリプト側に問題があったのではなくデータファイル側に問題があったのだ。何度も説明しているとおり、awkはデータを1行ごとにとってきてはそれをスクリプトに沿って処理する。ところがデータファイルcompacct.logを気をつけてよーく見てみると、最後の行に改行が入っていない。このためawkが「あー疲れた、でももう少しだ。もうひとガンバリするか」てな具合に最後の行を取りにいこうとした時に「もーかなわんなぁ。この行、改行コードがないままファイルの終わりになってるがな。こんなんレコードとして

認められへん！ぶーぶー

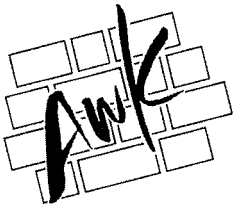
といった感じでawkが文句を言ったわけなのだ。

これを直すには最後の行に改行コードを入れてしまえばよいのだが、こうすると開始時間と終了時間を1行にまとめることができなくなってしまう。開始時間も終了時間も1行になってしまうのだ。shutdownコマンドさえ忘れずに必ず行うのであれば、開始時間と終了時間が交互に並ぶので利用時間の算出は簡単にできるのであるが、残念ながら人間はそれほど正確に動かない。これは図1-8を見てもわかる。こうした場合、どの行が開始時間でどの行が終了時間だかわからなくなってしまう。これでは集計してもとても信用のおけるデータとはいいがたい。

本当はcompacct.logを作成する時に、もっときちんとした形にするようなスクリプトを用意すればよいのであるが、話が一気に複雑になってしまうので、今回はこの、

パンツの紐が切れた

ような気持ちの悪い状態のまま、完成とする。



もうアプリなんていらない！

例題が１つでは寂しいし、この後の章の辛口の説明についていけるか心許ないので、もう１つawkを利用した例題を紹介することにしよう。今度の例題では、awkを利用して住所録を管理してみる。

DBMSよりawkだっ

一時期、処理スピードが重く操作が煩雑な日本語ワープロを使わずに、機能が単純で処理スピートの速いエディタを使って文章を起こすのが流行したことは記憶に新しい。これと同じようなことが、住所録等の個人レベルのデータベースにもいえる。

数千、数万のオーダーのデータを複数リレーションさせて動かすDBMS（データベース・マネージメントシステムね）は確かに強力なツールなのであるが、個人レベルの数十からせいぜい数百レコードのデータベース、しかも特にリレーションを組むこともない単純なものであれば、

何も高価なDBMSを使わなくても

エディタとfind,grep,awk等の検索ツールがあれば、ほとんどの処理はこと足りてしまう。しかも、これらのMS-DOSの標準ツールともいえるコマンド群は、DBMSに比べてオープンな環境を提供するので、ちょっと変わったことをしようとした時に、**自由度が高い**という点も見逃せない。

しかも、よしんばデータをDBMSにもっていかなければならなくなっても心配はいらない。ほとんどのDBMSがCSV（カンマ・セパレーテッド・バリュー）形式のテキストデータをサポートしているので、比較的簡単にデータをポーティングすることも可能だ。

住所録ファイルを作る

それでは、awkを使って住所録管理スクリプトを作る前に、肝心の住所録ファイルを作っておくことにしよう。ここで扱う住所録ファイルは、次に示すように各フィールドをコロンで区切って一件のデータ（つまり、1レコードということ）が1行になるようにする。

かな氏名:漢字氏名:郵便番号:住　所:電話番号:勤務先:備　考

リスト1-7が住所録ファイルのサンプルだ。これを見るとわかるとおり、少々見た目が悪くても各項目の桁を揃えるために余分なスペースを入れないほうが後々便利だ。またわからない項目は、無理に埋めずに空けておいて構わない。これを参考に各自の住所録ファイルをエディタなどを使って作成してみてほしい。ファイル名は“meibo”だ。

リスト1-7　住所録ファイルのサンプル

```
ふじ　えいじ:藤　英二::東京都葛飾区柴又 XX-X-XX:03-3:(宗)明日来教団:
しむら　たく:志村　拓:176:東京都練馬区豊玉北 X-X:3X-XX:世界征服(有):
なかべ　ただし:中部　正:184:東京都町田市柄割 X-X:3XX-XXXX:(株)印食:
おかべ　かず:岡部　和::大阪市大阪市雨野宝前寺 X-X:XX-XX:(株)トラ吉:
```

この書式、ある一点を除いて特に何も考えてない。順序と各項目の区切り文字さえ決まれば何でもよいのである。ある一点とは、平仮名による名前を一番最初の項目にした点だ。これは、sortコマンドを利用して住所録ファイルを名前順に並べ替える時のためだ。awkは、こうしたソート処理ができないわけではないのだが、苦手で非効率的なのだ。awkを使って苦しみながらソートするくらいなら、

フツーsortコマンドでしょ

となるわけだ。まぁsortコマンドのオプションを使えば、ソートする対象となる文字列の位置を指定できるのだが、そんなこと覚えなくてもソートの対象となる項目を一番最初に持ってきて、何も気にせずにsortコマンドにぶち込めば名前順にソートされる、といった経緯なわけだ。

```
A>type meibo | sort > meibo     ASCII MS-DOS TOOLS の sort
                                を使って"sort meibo  > meibo"
                                とはしないこと、もしやると...

A>
```

図1-9　名簿ファイルをソートする

整形してみよう

最初は小手調べに名簿ファイルを整形してみることにしよう。これは、たとえば年賀状を出す時などにプリンタとタックフォーム（あの、無地のシールのことネ）を使って出す場合に利用すると便利だ。

リスト1-8　名簿ファイルを整形する <format1.awk>

```
#
# 名簿ファイルを整形して出力する
# Copyright (c) 1993 by Taku-SHIMURA

BEGIN {
        FS = ":"
}
{
        print ""
        print ""
        print "〒" $3
        print $4
        print ""
        print $2 " 様"
}
```

リスト1–8が名簿ファイルを整形するスクリプトだ。これを実行すると、図1–10のようにあの名簿ファイルから綺麗な出力が得られるわけだ。

```
A>jgawk -f format1.awk meibo

〒
東京都葛飾区柴又 XX-X-XX

藤　英二 様

〒176
東京都練馬区豊玉北 X-X

志村　拓 様

〒184
東京都町田市柄割 X-X

中部　正 様
      :
      :
```

図1–10 <format1.awk>の実行例

それでは、リスト1–8について詳しく説明していくことにしよう。5行目の“BEGIN”は、前にも少し説明したが、ファイルを読み込む前に実行される命令を表す条件だ。

6行目がファイルを読み込む前に実行される命令なのだが、ここでは“FS”という変数に“：”という文字を代入している。FSはフィールドセパレータと呼ばれるシステム変数でフィールドの区切り文字を定義するものだ。最初の例題で説明したとおり、通常はフィールドの区切りはスペースかタブになっているが、このシステム変数FSの値を変更することでその区切り文字も変更することが可能なのだ。この変数の今回のmeiboファイルは各項目の区切りが“：”になっているので、これをFSに代入しているというわけだ。これでmeiboファイルの各項目は、図1–11に示すように各フィールドを表す変数に格納される。

```
かな氏名:漢字氏名:郵便番号:住　所:電話番号:勤務先:備　考
<-------------------------$0------------------------->
<--$1--> <--$2--> <--$3--> <-$4-> <--$5--> <-$6-> <-$7->
```

図 1–11 住所録ファイルのデータフォーマット

8行目の“{”と15行目の“}”で囲まれた命令は、8行目に何の条件もないので、すべてのレコードで実行される。これは見ればわかるとおりmeiboファイルの各項目を“print”命令を使って画面に表示しているだけだ。ここで注意しなければならない点は、9、10、13行目の“print ””””という命令で、これは空行を出力するために使用されている。「何も出力しなくていいなら“print”と書くだけでもいいじゃないの？」と思って単に“print”だけを書くと、図1–12のようになってしまう。

```
A>jgawk -f format2.awk meibo
ふじ　えいじ:藤　英二::東京都葛飾区柴又 XX:XX-XX:(宗)明日来教団:
ふじ　えいじ:藤　英二::東京都葛飾区柴又 XX:XX-XX:(宗)明日来教団:
〒
東京都葛飾区柴又 XX-X-XX
ふじ　えいじ:藤　英二::東京都葛飾区柴又 XX:XX-XX:(宗)明日来教団:
藤　英二 様
しむら　たく:志村　拓:176:東京都練馬区豊玉北 X-X:XX-XX世界征服:
しむら　たく:志村　拓:176:東京都練馬区豊玉北 X-X:XX-XX世界征服:
〒176
東京都練馬区豊玉北 X-X
しむら　たく:志村　拓:176:東京都練馬区豊玉北 X-X:XX-XX世界征服:
志村　拓 様
なかべ　ただし:中部　正:184:東京都町田市柄割 X-X:XXXX(株)印食:
なかべ　ただし:中部　正:184:東京都町田市柄割 X-X:XXXX(株)印食:
〒184
東京都町田市柄割 X-X
なかべ　ただし:中部　正:184:東京都町田市柄割 X-X:XXXX(株)印食:
中部　正 様
　　:
　　:
```

図 1–12 <format2.awk>の実行

つまりprint命令は、何も指定しなかった場合その時のレコードの内容"$0"を出力してしまうのだ。空行を出力したい場合は、必ず"print """と指定するようにすること。

検索してみよう

今度はもう少し複雑な処理をしてみよう。名前で検索してその人のデータを表示するスクリプトだ(**リスト1-9**)。スクリプトを説明する前にどんな感じで実行されるか見てみることにしよう。**図1-13**がfindnam1.awkの実行例だ。スクリプトを実行すると、対話形式で名前を聞いてくるので、検索したい人の名前を入力するとawkは名簿ファイルを検索してその名前の含まれる行を整形して表示する。

リスト1-9　名簿ファイルを検索する <findnam1.awk>

```
#
# 名簿ファイルから名前で検索する
# Copyright (c) 1993 by Taku-SHIMURA

BEGIN {
        FS = ":"
        printf "名前を入力してください: "
        getline name < "/dev/stdin"
}

$2 ~ name {
        print ""
        print "名　　前: " $2
        print "ふりがな: " $1
        print "郵便番号: " $3
        print "住　　所: " $4
        print "電話番号: " $5
        print "勤 務 先: " $6
        print "備　　考: " $7
}
```

＜実行結果＞

```
A>jgawk -f findnam1.awk meibo
名前を入力してください: 志村          ←「志村」で検索
名　　前: 志村　拓                    ←漢字氏名に「志村」が含まれる
                                        レコードが整形されて表示される。
ふりがな: しむら　たく
郵便番号: 176
住　　所: 東京都練馬区豊玉北 X-X
電話番号: 03-39XX-XXXX
勤 務 先: 世界征服(有)
備　　考:

A>jgawk -f findnam1.awk meibo
名前を入力してください: 遠藤          ←「遠藤」で検索
                                        ←「遠藤」が見つからないので、
                                            何も表示されない
A>
```

図 1-13 <findnam1.awk>の実行

それではスクリプトの説明をすることにしよう。大半の部分は前のスクリプトで使った命令なので、ここでは新たに使った

3つの命令について説明する。

最初の新命令は、7行目の“printf”だ。いままで使っていた“print”に“f”がついただけのものなので、間違えないように気をつけること。この“printf”は、C言語の同名の関数と互換性のある命令で、書式指定のできる出力命令だ。C言語を知らない向きは、改行の行われない“print”と考えてくれればよい（たいぶ違うケド）。

次の8行目にも新しい命令が使われている。“getline name <”/dev/stdin”” は、キーボードから入力された文字列を変数nameに代入する命令だ。正確に言うと“getline”命令はデータファイルから1レコード分のデータを取得して、その後に書かれた変数に代入するものなのだが、今回の例では“getline name”の後にデータの取得先を変更するリダイレクション命令“<”/dev/stdin””が付いているので、データファイル、つまり名簿

ファイルから1レコードを変数nameに読み込まず“/dev/stdin”から読み込むようになったわけだ。この“/dev/stdin”というのは

”標準入力”

と呼ばれるもので、キーボードからの入力ということになるのだ。

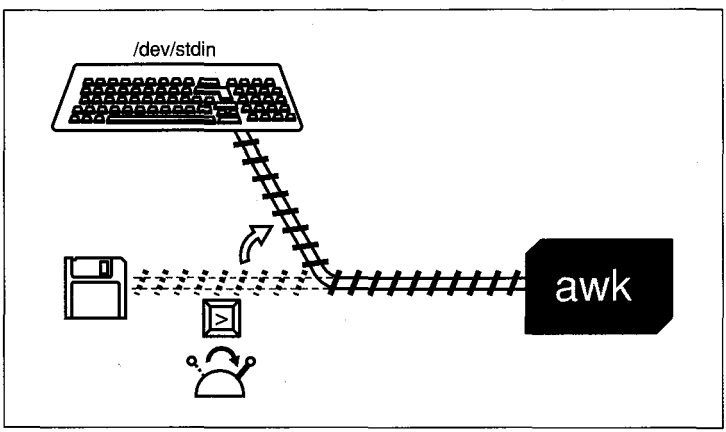


図1-14 標準入力“/dev/stdin”へのリダイレクト

ちなみに、この“/dev/stdin”の代わりにファイル名を指定するとそのファイルから1レコードのデータを取得する。まぁ、ここでは話を難しくしたくないので、これ以上の説明はしないことにする。とにかくスクリプトの途中に対話形式でキーボードから文字列を入力させたいような場合には、“getline”命令をこのように

<”/dev/stdin”と組み合わせればよい

ということだけ覚えておいてほしい。

最後の新命令は、11行目の“$2 ~ name {”だ。この書き方の意味は、「$2の中にnameの文字が含まれていた場合、続く命令を実行する」とい

うものだ。“$2 == name {”でもよかったのだが、この書き方だと検索する名前として入力した文字列が、名前の項目に完全に一致しないと検索されないことになり非常に不便なので、“$2 ~ name {”を使ったわけだ。

このスクリプトの検索をもっと柔軟にするためには、11行目を“($1 ~ name) ||($2 ~ name) {”とすればよい。“||”は「または」という意味で、この記述だと「$1の中にnameが含まれているか、または$2の中にnameが含まれている場合に続く命令を実行する」ということになる。$1は「かな氏名」で、$2は「漢字氏名」だから、11行目をこのように書き直すと、「かな氏名」でも「漢字氏名」でも検索ができるようになる。

検索スクリプトの改良

検索部分の改良案も出たし、名前が見つからなかった場合に何のメッセージも表示せず無愛想な点も気にいらないので、このスクリプトを改良してもう少し見栄えのいいものにした。**リスト1-10**がスクリプトで、**図1-15**がその実行結果だ。

リスト1-10 名簿ファイルを検索するスクリプト改良版 <findnam2.awk>

```
#
# 名簿ファイルから名前で検索する(改良版)
# Copyright (c) 1993 by Taku-SHIMURA

BEGIN {
        FS = ":"
        printf "名前を入力してください: "
        getline name < "/dev/stdin"
 }

($1 ~ name) || ($2 ~ name) {
        count = count + 1
        print ""
        print "名     前: " $2
        print "ふりがな: " $1
        print "郵便番号: " $3
        print "住     所: " $4
        print "電話番号: " $5
        print "勤 務 先: " $6
```

```
        print "備     考: " $7
}

END {
        print ""
        if(0 == count) {
                print name "という名前は見つかりませんでした。"
        } else {
                print "以上" count "件のデータが見つかりました。"
        }
}
```

<実行結果>

```
A>jgawk -f findnam2.awk meibo

名前を入力してください: 志村

名     前: 志村  拓
ふりがな: しむら  たく
郵便番号: 176
住     所: 東京都練馬区豊玉北 X-X
電話番号: 03-39XX-XXXX
勤 務 先: 世界征服(有)
備     考:

以上1件のデータが見つかりました。

A>jgawk -f findnam2.awk meibo
名前を入力してください: しむら        ←かな氏名でも検索が可能になった。

名     前: 志村  拓
ふりがな: しむら  たく
郵便番号: 176
住     所: 東京都練馬区豊玉北 X-X
電話番号: 03-39XX-XXXX
勤 務 先: 世界征服(有)
備     考:

以上1件のデータが見つかりました。
```

```
A>jgawk -f findnam2.awk meibo
名前を入力してください: 遠藤

遠藤という名前は見つかりませんでした。    ←名前が見つからなかった場合
                                          メッセージが出力されるように
                                          なった
```

図 1–15 <findnam2.awk>の実行

先に指摘した 11 行目の条件以外の主な改良点を簡単に説明しておこう。まず、12 行目の“count = count + 1”だが、これは 11 行目の条件にあったレコード、言い換えれば指定された名前に一致したデータの件数を数えるためのものだ。

23 行目から 30 行目は、条件として END が指定されている。END の条件は前にも説明したとおりデータファイルをすべて読み終わった後に実行されるというものだ。END の条件で実行される 24 行目から 29 行目までの命令は、変数 count が 0 の時に 26 行目の命令が 0 以外の時に 28 行目の命令が実行される。

住所から検索

先のスクリプトをチョコチョコっと変更して、住所から検索するスクリプトを作ってみた (**リスト 1–11**)。内容は先のスクリプトと 83 %同じで、異なる部分も主にメッセージや変数名の変更だけなので、このスクリプトに関する説明は特に行わない。

リスト 1-11 名簿ファイルを住所で検索 <findadr1.awk>

```
#
# 名簿ファイルから住所で検索する
# Copyright (c) 1993 by Taku-SHIMURA

BEGIN {
        FS = ":"
        printf "住所を入力してください: "
        getline addr < "/dev/stdin"
}

$4 ~ addr {
        count = count + 1
        print ""
        print "名     前: " $2
        print "ふりがな: " $1
        print "郵便番号: " $3
        print "住     所: " $4
        print "電話番号: " $5
        print "勤 務 先: " $6
        print "備     考: " $7
 }

 END {
         print ""
       if(0 == count) {
          print addr "という住所は見つかりませんでした。"
         } else {
          print "以上" count "件のデータが見つかりました。"
         }
 }
```

このスクリプトも結構便利だ。たとえば東京都に住んでいる人のリストを作ることも、このスクリプトを使えば、

一撃だ!

実際に行ってみたのが図 1-16である。

```
A>jgawk -f findadr1.awk meibo
住所を入力してください: 東京都

名　　前: 藤　英二
ふりがな: ふじ　えいじ
郵便番号:
住　　所: 東京都葛飾区柴又 XX-X-XX
電話番号: 03-3
勤 務 先: (宗)明日来教団
備　　考:

名　　前: 志村　拓
ふりがな: しむら　たく
郵便番号: 176
住　　所: 東京都練馬区豊玉北 X-X
　　:
　　:
名　　前: 工藤　しのぶ
ふりがな: くどう　しのぶ
郵便番号: 541
住　　所: 京都府東京都市宇曽 5-16-3
電話番号: 0522-43-XXXX
勤 務 先:
備　　考:

以上14件のデータが見つかりました。
```

図 1–16 <findadr1.awk>の実行

ゲッ!

東・京都も東京都として検索されてしまった。ま、漢字で書けば同じ「東京都」になるわけだから、それをコンピュータに判断しろというのも酷というものだ。

「だったら、せめて住所の先頭が"東京都"で始まるものといった条件が与えられれば」。確かにそのように指定できれば東京都在住者のリストを作ることができるだろう。できるんですよ、こうした指定が。図 1–17を見てほしい。

```
A>jgawk -f findadr1.awk meibo
住所を入力してください: ^東京都

名　　前: 藤　英二
ふりがな: ふじ　えいじ
郵便番号:
住　　所: 東京都葛飾区柴又 XX-X-XX
電話番号: 03-3
勤 務 先: (宗)明日来教団
備　　考:

名　　前: 志村　拓
ふりがな: しむら　たく
郵便番号: 176
住　　所: 東京都練馬区豊玉北 X-X
　　:
　　:
名　　前: 中部　正
ふりがな: なかべ　ただし
郵便番号: 184
住　　所: 東京都町田市柄割 X-X
電話番号: 03-39XX-XXXX
勤 務 先: (株)印食
備　　考:

以上13件のデータが見つかりました。
```

図 1-17 <findadr1.awk>の実行

住所として“^東京都”と指定しているが、この文字列の先頭の“^”が項目の先頭を表している。つまり“^東京都”は、「項目の先頭が東京都」ということになる。

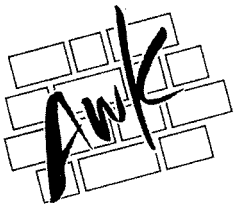
この他にも、“大阪府|京都府”と指定すると、「大阪府または京都府」ということになる(図 1-18)。

```
A>jgawk -f findadr1.awk meibo
住所を入力してください: 大阪府|京都府

名　　前: 岡部　和
ふりがな: おかべ　かず
郵便番号: 610
住　　所: 大阪府大阪市雨野宝前寺 XXX-X
電話番号: 06-XXX-XXX
勤 務 先: (株)トラ吉
備　　考:

名　　前: 工藤　しのぶ
ふりがな: くどう　しのぶ
郵便番号: 541
住　　所: 京都府東京都市宇曽 5-16-3
電話番号: 0522-43-XXXX
勤 務 先:
備　　考:

以上2件のデータが見つかりました。
```

図 1–18 <findadr1.awk>の実行

これらの“^”や“|”を含んだ文字列は、「正規表現」と呼ばれるものだ。この他にも正規表現には、さまざまな指定方法があり、これらを使用すれば、DBMS よりも断然多彩な検索が可能になる。

リスト 1–11の 11 行目の“$4 ~ addr {”は、「$4 に addr が含まれている場合」ではなく、実は「$4 に addr で指定した正規表現と一致する部分がある場合」という意味なのだ。



正規表現うわー!

正規表現というのは、ある文字列を特定するためのルールだ。何かとても難しそうに聞こえるがそんなことはない。MS-DOS にはファイル名を適当に指定するためのワイルドカードと呼ばれるものが用意されているが、正規表現はこれをもっと多彩にしたものと思えばよいのだ。

メタ文字とは

MS-DOS のワイルドカードは、"?" が任意の 1 文字を、"*" がそれ以降の文字列は何でよいといった意味になっているが、正規表現にも同じように特殊な意味を持つ文字がある（**表 1-1**）。これらの文字列をメタ文字と呼ぶ。このメタ文字を利用した表現を 0 個以上含む文字列を正規表現という。

メタ文字	意　味
.	任意の 1 文字に一致
*	前の文字の 0 個以上の繰り返しに一致
+	前の文字の 1 個以上の繰り返しに一致
?	前の文字がないか、1 個あるかに一致
^	行頭に一致
$	行末に一致
¥	エスケープキャラクタ

表 1-1　正規表現で使用するメタ文字

気をつけなければならない点は、ワイルドカードの "?" と "*" と、正規表現の "?" と "*" が違う意味になっている点だ。**表 1-2** を見て、よーく考えれば理解できると思うが、ワイルドカードの "?" は正規表現では "." に、

“*”は“.*”と書く。

“*”は一人では生きていけない

特に初心者が間違えやすいのが、“*”だ。“*”,“+”や“?”は、前の文字のグルになって動く。たとえば、“うわー*”と書くと、正規表現を解釈する側（つまり、この場合awkであるが）は、“うわ”と“ー*”と認識される。最初の“うわ”はそのままの文字ということになる。その次に続く“ー*”は、「tの文字の0個以上の繰り返し」という意味になる。つまり、“ー*”の指定では、“ー”が0個、つまり“ー”がなくてもよいということになり、この“うわー*”という正規表現では、“うわー”、“うわーーーー”はもちろんのこと、“うわ”にも一致してしまうのだ。

うわ、うわー、うわーーー！！

ワイルドカードのイメージが強いと、“*”は、それ単独で意味を持つものと考えがちで、このことから「前の文字の0個以上の繰り返し」という意味が理解しにくいものとなってしまうのだ。“*”, “+”, “?”は前の文字とあわせて正規表現となる点を意識してほしい。

正規表現	一致する内容
^#	C言語のプリプロセッサの命令に一致
ばかやろー+	ある映画監督の雄叫びに一致
を256倍使うための本$	蘊蓄ばかりであまりためにならない書名に一致
¥x1A	CP/Mの呪縛コードに一致

表1–2　正規表現の使用例

ばかやろーを256倍使うための・・・えっ?

最長一致うわーっ

先の説明で「はて?」と疑問を感じた人は鋭い。たとえば“うわー?”という正規表現で“うわーーーっ!”に一致させた場合では、この文字列のどの部分に一致するのだろう?“うわー*”の正規表現は“うわ”にも“うわー”にも“うわーー”にも“うわーーーーー”にも一致するはずだが…

awkの正規表現では(というより、フツー正規表現と言えば、こうなのだが)、最長一致というルールがあり、これに沿って文字列の一致を行う。最長一致とは、指定した正規表現で一致することのできる

もっとも長い文字列と一致する

という意味だ。つまりこの例の場合、正規表現“うわー*”は、“うわーーーっ!”の中の“うわーーー”の部分に一致することになるわけだ。

正規表現で不便!?

「正規表現は、便利、便利」とうかれていてはならない。正規表現がついているがために面倒なことになる場合もあるのだ。たとえばワイルドカード“*.*”が使用されている部分を検索したい場合などがそれだ。“*”も“.”もメタ文字なので、そのまま指定しても“*”や“.”という文字の意味にならないのだ。ここで登場するのが、**‘¥’エスケープキャラクタ**だ。

メタ文字の前に“¥”を付けると、その文字はメタ文字として扱われず、通常の文字として扱われるようになる。たとえば、先の例だと、**“¥*¥.¥*”**としてやれば、**“*.*”**を検索することができる。もちろんエスケープキャラクタ自身の“¥”という文字を検索したい場合も**“¥¥”**としてやらなければならない。たとえば、C言語のソースのなかから改行コードを示す“¥n”を検索したい場合は、**“¥¥n”**としてやらなければならないのだ。

エスケープキャラクタにはメタ文字の意味を殺す以外に、その後ろに指定した文字を書くことによって制御キャラクタを表す機能がある。これを

エスケープシーケンスと呼ぶ。

エスケープシーケンス	意　味
¥a	ベル文字 (いわゆる Ctrl+G)
¥b	バックスペース文字
¥f	フォームフィード文字
¥n	ニューライン文字
¥r	キャリッジリターン文字
¥t	タブ文字
¥v	垂直タブ文字
¥<oct>	<oct>を8進数とする文字コード (<oct>は3桁まで)
¥x<hex>	<hex>を16進数とする文字コード
¥<chr>	上記以外の文字<chr>の場合は、<chr>の文字自体

表 1-3　エスケープシーケンス

続・メタ文字

話を簡単にするために、比較的簡単なメタ文字の説明を先に行ったが、実は、この他にも awk で使用できる正規表現のメタ文字がいくつか存在する。表 1-4に、それらのメタ文字を挙げてみた。

メタ文字	意　味
(re)	"("と")"で囲まれた正規表現 re をグループ化する
[ ]	"["と"]"で囲まれた文字列の中の任意の1文字に一致
[^]	"[^"と"]"で囲まれた文字列以外の文字に一致
re1|re2	正規表現 re1 または re2 に一致する

表 1-4　正規表現で使用するメタ文字

*正規表現は、英語で Regular Expression と書く。そこで正規表現を表す文字として、r や re が使われるケースが多い。ちなみに "grep" コマンドの "re" も正規表現を表す "re" であり、UNIX の ed/ex でもいいけど) エディタの "g/正規表現/p" からもじったものなのだ。

最初の "( )" は、よく "*" や "+" と組み合わせて使う。"(シク)+" とすれば、"シク", "シクシク"、"シクシクシク" といった文字列に一致させることができる。先に説明した "*", "+", "?" は、この "( )" と組み合わせることで、文字ではなく文字列や正規表現を対象とすることができるようになる。この表現を覚えると複雑な、ともすると、

自分でも予測しきれない

指定を行うことができるので、楽しい反面、十分に気をつけてほしい。

正規表現	意　味
(パチ)+	拍手に一致
(キューちゃん)*	あるオオムの独り言に一致
(ルー*)+	夜明けのスキャットに一致

表 1-5　"( )"の使用例

次の "[ ]" と "[^]" は非常に便利だ。たとえば "[abc]" と書くと、"a" または "b" または "c" という意味になる。逆に "[^abc]" と書けば "a" または "b" または "c" 以外の 1 文字という意味になる。

"[abc]" は "[a-c]" と書き直すこともできる。これは a から c までの文字という意味だ。この程度の例ではその利便性を認識できないかもしれないが、たとえばアルファベットすべてに一致する表記をする場合、この "-" を使い "[a-zA-Z]" と記述できる ("-" を使用しないとどのような状態になるかは、想像がつくだろう)。もちろん "-" は "[^]" でも利用可能で "[^a-zA-Z]" と書けばアルファベット以外の文字という意味になる。

ちなみに "[c-a]" と書いても "[abc]" と同じ意味にはならないので注意すること。"-" は、前に書いた文字コードから後ろに書いた文字コードま

での文字を指定したことになる。ここで言う文字コードとは半角文字であればJIS X0201を(簡単に言えばASCIIコードに半角カナが追加されたもの)、全角であればJIS X0208(いわゆるJIS漢字コード)を指す。つまり"-"の前に書く文字のコードの方が後ろに書く文字のコードより小さくなくてはうまく指定できないわけだ。このあたりの話は少々難しいが、一度ゆっくりと、**JISコード表とニラメッコ**してみることをお勧めする。

正規表現	意　味
[A-Za-z_][A-Za-z0-9_]*	C言語の関数または変数名に一致
^#[¥t]*include[¥t]+[<"][^<>"]*[">]	C言語のinclude文に一致
[+-]?[0-9]+¥.?[0-9]*	数値に一致
(&[Hh]|0[xX])?[0-9A-Fa-f]+	16進数に一致
[亜-腕]	第一水準漢字一文字に一致
[弌-龠]	第二水準漢字一文字に一致
[亜-龠]	任意の漢字一文字に一致 (JIS X0212は忘れてください)
[ア-ヴ]+	よく判らないコンピュータ用語に一致
[A-Za-z]¥.[A-Za-z]¥.[A-Za-z]¥.	もはや、想像もつかない頭文字系コンピュータ用語に一致

表1-6　"[ ]"を使用した正規表現

さて、"はい"または"いいえ"に一致させる正規表現を書くには、どうしたらよいだろう。今までに紹介した正規表現で無理やり書くとすれば、"[はい]いえ?"といったところだろうか。この表記では

"はいえ"のような怪しげな京都弁にも

一致してしまう。このような場合に便利なのが"|"だ。

"はい|いいえ"は、"はい"または"いいえ"の意味になるのだ。"|"の前後の文字列には、正規表現も記述できるので、"うそー*|ほんとー*"と

いった正規表現もできる。ちなみに、この場合“(うそ|ほんと)ー*”と書くこともできる。また“|”は並べて複数個書いても構わない。“(うそ|ほんと|まじ)ー*”と書けば“うそ”にも“ほんとー”にも“まじーーーー”にも一致させることができる。

正規表現	意　味
(バキュ|パ|バ)ン	銃声に一致
(キュル|キョ)+	土曜日の大垂水峠の騒音に一致
(だしぃ+|でぇ+)[、。]	アーパーねーちゃんの会話に一致

表 1-7　“|”を使用した正規表現

正規表現の思わぬ落とし穴

ここまで読んできて、正規表現で使用されるメタ文字とその意味がおぼろげながら理解できたのではないだろうか。あとは現実のさまざまな局面でどんどん利用していけば、短期間に正規表現をマスターすることができるはずだ。最後に

正規表現の落とし穴をお教えしておこう。

実は正規表現には、いくつかのレベルというか、種類が存在している。たとえば grep で使用できる正規表現は、ここで説明した awk で使用できる正規表現のうち“+”や“?”といった表現はできない。また、vi や sed などでは awk のものに加えて語句境界を表す“¥<”や“¥>”といった表現、置換の際に便利なタグ付き正規表現などが追加されている。正規表現の世界は結構深い部分もあるので、ここまで読んで awk の正規表現が理解できたと思って「あぁ、正規表現ならまかせてヨ」などと威張っていると足もとをすくわれることもありうるので、注意すること。

awkの変遷について

◇ ◇ ◇

<awkの誕生>

awkはUNIXの世界から生まれた言語(?)である。最初のawkは1977年に開発された。Aho、Weinberger、Kernighanの3人により作られ、彼ら3人のイニシャルを集めてawkと呼ばれた。これがその名の由来である。

たぶん“Version7 Unix”と一緒に配布されたと考えられる。これが元祖のawkであり、初期のUNIXユーザーのバイブルとも言える『UNIXプログラミング環境』(The UNIX Programming Environment)で紹介されているのもこれだ。

<awk → nawk → gawk → jgawk>

また、1987年にその言語仕様を拡張しAT&TのSystem–V Release3.1に付属されたawkもある。これは俗にnawkと呼ばれるもので、ユーザー定義関数や組み込み関数などが新たにサポートされたようだ。

FSF(フリーソフトウェアファウンデーション)ではGNUプロジェクトの一環として1986年にgawk(GNU awk)を発表した。これはnawkの機能をすべて含んでいるほか、細かい拡張がなされている。また、MS–DOS上にも移植され、PC上でも使えるようになった。gawkはGNUが定める配布規定である“copyleft”に則って配布されている。

もともと文字が26種類しかない人たちの考えたものなので、gawkは日本語に対応していない。しかし田中良知(serow)氏の尽力により日本語化され、jgawkとして現在私たちが使っているものとなった。

awkの入手法について

◇ ◇ ◇

<j?gawk の入手方法>

gawk は、GNU が定める配布規定である “copyleft” に従って配布されている。そのため入手の際に代金を払う必要はない (FSF で寄付は受け付けている)。逆にどこかのショップに行って買うというわけにもいかないわけだ。

internet に接続できる人は、ftp サイトから持ってくるのが最も望ましい。しかし大多数の internet にアクセスできない人達には…次のような方法がある。

- ASCII–NET などのパソコン通信からダウンロードする
- 友人などにコピーしてもらう
- 雑誌などの付録ディスクから入手する
- FSF に直接依頼して、テープもしくは CD–ROM を送付してもらう

などである。まあ月刊アスキーなどのバックナンバーを調べて付録ディスクを見つけるのが最も現実的かも知れない。

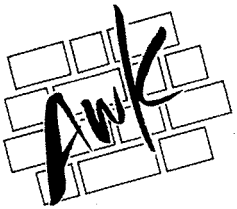
なおバージョンが古いと動かないスクリプトもあるので、入手する際には十分注意してもらいたい。

<バージョンアップについて>

gawk は現在でもバージョンアップを続けており、本書の執筆時点での最新バージョンは “2.15.2” である。これに対応する “2.15.2 + 日本語対応パッチ Ver1.0” が jgawk の最新のものと言える。 (1993/07/18 現在)

awkを解剖する!

1章では、ネコをかぶって比較的やさしく書いたつもりだが、awkがいったいどんなものかご理解いただけただろうか？本章では、1章で完璧には説明できなかったawkの言語仕様を、なるべく体系立てて説明する。「awkの言語仕様なんて、知ってるよ」などという人にかぎって、マニュアルの記述にだまされていてその実を知らないケースは少なくない。もう1度初心に戻って本章を読み、目の鱗を落としてほしい。



awkの背骨

いくらawkには使い捨てのスクリプトが多いといっても、言語仕様を知らないまま適当にスクリプトを書いたりしてはならない。少なくとも本章で説明するようなことくらいは知った上でスクリプトを書かなければ思わぬところでawkの特異な仕様にはまってしまうこともあるだろうし、第一awkの作者に対して失礼というものだ。

そこで本章では、awkの基本的な仕様を説明するとともにポイントごとに注意が必要な部分を逐次説明していくことにする。

awkスクリプトの骨格

awkのスクリプトの構成要素は、「ルール」と呼ばれている。awkのスクリプトは、1つ以上のルールによって構成されているのだ。ルールは、「パターン」と「アクション」と呼ばれる2つの要素によって構成されている。こんな風に書くと難しそうに見えるが、実際に読んでもらえばどうと言うことはない。ひとまずここでは各部の名称を覚えてもらいたい。

```
<パターン> { <アクション> } ←これがルール
<パターン> { <アクション> }
        :
        :
<パターン> { <アクション> }
```

アクションの部分は多くの場合、複数の命令を書く際に1命令1行の形で書く。これは改行を命令の区切り文字としてawkが認識するからだ。命令の区切り文字としては、この改行の他に、";"が使用できるようになっている。たとえばコマンドラインにスクリプトを記述する際は、こちらの

";" を使用すれば複数の命令を1行に記述することができる。ただ前の章でも説明したとおり、MS-DOS の COMMAND.COM は機能がチープなので、この方法を使っても難しいかもしれない。

```
<パターン> {
    <命令>          スクリプトファイルを記述する場合は、
    <命令>          このような形で1つのルールを複数行に
      :             わたって記述する場合が多い
      :
    <命令>
}

<パターン> { <命令>; <命令>; ... <命令> }

                コマンドラインからスクリプトを指定する場合は、
                このように";"を命令のセパレータとして使用して、
                1行にする場合が多い
```

ここで注意しなければならないのは、"{" は必ずパターンと同じ行に書かなければならない点だ。もし、パターンとそのアクションの括る "{" の間に改行を入れてしまうと別の意味になってしまうからだ。これは後の方で説明するが、ルールのパターン及びアクションは、そのいずれか一方を省略可能な仕様になっているので、2つのルールになってしまうのだ。

```
<パターン> {
    <命令>          こちらは、1つのルールであるが、下の
    <命令>          ルールは、
      :
      :
    <命令>
}

<パターン>          これが1つのルール
{                   そして、こちらがもう1つのルールとい
    <命令>          った具合に、2つのルールとなってしま
    <命令>          う
      :
      :
    <命令>
}
```

ちなみに、「ルール」という言葉は、gawkのマニュアルに書かれている言い回しで、awkの「教典」と呼ばれている“The AWK Programming Language”では、「パターン–アクション文」という言い回しが使用されている。また、awkの故郷であるUNIXのオンラインマニュアルも、「パターン–アクション文」という表現が一般的のようだ。まぁ、これは「教典」の著者たちがUNIXに深く関与していることを考えれば当然のことであろう。

しかし本書では、書かれている内容の多くがgawkに主眼を置いていること、「パターン–アクション文」という表記では混乱を招く可能性があること等を考え「ルール」を使用することにする。

awkの基本動作

awkの基本動作は、

おかずの品数が多い「ゴーヂャスな食事」に似ている。

awkで処理するデータを「ごはん」、ルールを「おかず」とすると、ごはん一膳を手に、おかずを端から1つずつ食べていき、最後のおかずを食べ終わるとごはんをおかわりして最初のおかずから繰り返すといった感じだ。

awkがデータという「ごはん」を食べるとき、一度に食べる量は決まっている。この一度に食べる量が「1レコード」という単位だ。1レコードはデフォルトの状態では1行になっている。一方awkの「おかず」であるルールを食べるということは、そのルールを実行することに相当する。

つまり、awkはデータファイルから1レコード分のデータを取得してはスクリプトの先頭から順にルールを実行していく（ルールの実行とはパターンを評価し、それが真である場合にアクションを実行することだ）。レコード、パターン及びアクションの説明は後で説明するのでご心配なく。

最後のルールを実行した後、awkはデータファイルから次の1レコードを取得し、再びスクリプトの先頭から順にルールを実行する。これをデータファイルから読むべきデータがなくなるまで繰り返すというのが、awk

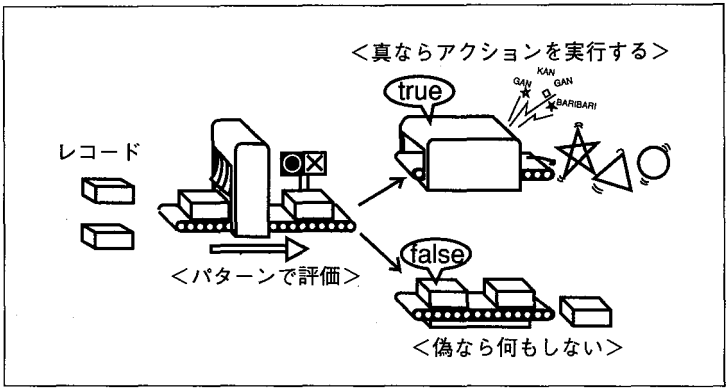


図 2-1　パターン to アクション

の基本動作だ。

さて、この一番外郭の基本動作をコントロールする制御構造として、next と exit という命令が用意されている。これらの命令はアクションのなかに記述する。

- **next 命令**　それ以降のアクション及びルールの実行を中止し、データファイルから次のレコードを読み込み、最初のルールに制御を移す。
- **exit 命令**　それ以降のアクション及びルールの実行を中止するのに加え、それ以降のデータの読み込みをも中止しスクリプトを終了する。ちなみに後で説明する END パターンのルールがある場合は、END パターンのアクションを実行する。

パターンの役割

awk は、ルールという「おかず」を食べる時、いつもすべての「おかず」を食べるわけではない。「ごはん」であるデータの内容によって、食べる「おかず」と食べない「おかず」をセットできるのだ。たとえばカレーという「おかず」は「ごはんが白いごはんやバターライスの時は食べるが、そ

れ以外のお粥やチャーハンの時は食べない」という風に定義することができる。

この「ごはん」(つまりデータ)によって「おかず」(つまりルール)を食べるが食べないか(つまり実行するか否か)を決めるのが、ルールの「パターン」部なのだ。

おかず選ぶんじゃありませんっ!

パターンには多くの表記方法が用意されている、と言いたいところであるが、実はパターンとして使用できる表記方法は、以下に示す2種類しかない。「えー!そんなはずはない、"/正規表現/"といった形だってあるし、複合パターンだってあるじゃないか」と思う人があるかも知れないが、無理もない。なにしろ「教典」をはじめとする多くのawk解説書にそうした記述があるのだから。

```
式
式1,式2
```

一般的によく使用される"/正規表現/"といったパターン(これを照合パターンと呼ぶ)は、実は"文字列 ~ /正規表現/"という式の省略形であるとみなすことができる(正確に言えば照合演算子が照合パターンの派生なのかも知れない)。この"~"は照合演算子と呼ばれるもので、これも立派な式の1つである。ちなみにアクションのなかでもこの記述は使用できる。たとえば以下のようなスクリプトを実行すると、数字が含まれる行に限って出力する(まぁ、こうしたスクリプトを書く場合は、一般的には"/[0-9]/"と書くが)。

```
{if(/[0-9]/) print}
```

もちろんこの照合パターン自体も値を持ち、$0に照合パターンが見つかった場合は1を、見つからなかった場合は0を返す。

複合パターンにいたっては、単に複数の式をブール代数系の演算子でつなげただけのものであり、もしこれを式と区別するのであればこれらの演算子は式の構成要素でないことになってしまう。つまりこれも式と考えてなんら問題ないわけである。

さて、最初の表記方法であるがこれは式の計算結果が0以外、つまり真の場合そのルールのアクションが実行され、0つまり偽の場合アクションは実行されないことになる。

次の「式1,式2」といった書き方であるが、この場合式1が真になると式2が真になるまでアクションが実行されるようになる。一度式1が真になってしまうとその後式1が真であろうが偽であろうが、式2が真になるまでアクションが実行される点が面白い。

ちなみに

式2が真になった行ではアクションが実行されるが、次回からは式1が真にならない限りアクションは実行されない。これによってデータの特定の領域をアクションに処理させているわけだ。

このあたりは下記のような簡単なスクリプトで実験できるので、実際にやってみるとawkの仕様の健気さが見えてきて面白い。

リスト2-1　<act2.awk>

```
    { a = $1 + 0; b = $2 + 0; print "(" a "," b ")"}
a,b { print "a,b action" }
```

<実行結果>

```
A>jgawk -f act2.awk
0 1
(0,1)
0 1
(0,1)
1 1
(1,1)
a,b action
0 0
(0,0)
1 0
(1,0)
a,b action
0 2
(0,2)
a,b action
0 2
(0,2)
^C

A>
```

図 2-2　<act2.awk>の実行

最後になったが、ルールのアクションは省略することができる。アクションを省略した場合は式が真であるとみなされる。つまりすべてのレコードでアクションが実行されることになる。

BEGIN, ENDパターン

awk の食事は意外と豪華で、ちゃんと「ごはん」を食べる前の「オードブル」と、「ごはん」を食べ終わった後の「デザート」が存在する。

パターンには、“BEGIN” と “END” の 2 つの特殊なものがある。BEGIN パターンは、データの読み込みを始める前に一度だけ真となるパターン、逆に END パターンは、データの読み込みがすべて終わった時点で一度だけ真となるパターンだ。つまり、

BEGINパターンは「オードブル」
ENDパターンは「デザート」

ということだ。

多くの場合、BEGINパターンは各種初期化を行うために、ENDパターンは終了時の各種処理を実行するために使用する。また複雑な処理を行うスクリプトでは、通常のルールで読み込んだデータを一度変数に保持して最後にENDパターンで処理を行うようなことをする場合も少なくない。

ところで、

BEGINパターンやENDパターンを複数表記

することができるのを御存知だろうか。これらのパターンが複数指定されている場合は、スクリプトの先頭から順に実行される。

これは、awkの起動時の-fオプションを複数使用する方法とあわせて、awkスクリプトのライブラリを作成する場合に有効だ。たとえば16進数の文字列を数値に変換するような関数をライブラリ化するような場合、BEGINパターンのアクションで変換テーブルの作成を行うのが普通であるが、これを以下のような形で、1つのファイルにまとめてみる。

リスト2-2　16進数を変換するBEGIN <htoi.awk>

```
BEGIN {
    split("1,2,3,4,5,6,7,8,9", _HtoI, ",")
    _HtoI["0"] = 0
    _HtoI["A"] = 10
    _HtoI["B"] = 11
    _HtoI["C"] = 12
    _HtoI["D"] = 13
    _HtoI["E"] = 14
    _HtoI["F"] = 15
}
func htoi(str,  chr, loop, ret)
{
    ret = 0
```

```
    for(loop = 1; loop <= length(str); ++loop) {
        chr = toupper(substr(str, loop, 1))
        if(chr !~ /[0-9A-F]/) {
            break
        }
        ret = ret * 16 + _HtoI[chr]
    }
    return ret
}
```

この関数を使用するスクリプトでは、特に何も気にせずあたかもこの関数が組み込み関数であるかのように、ただ単純に使えばいい。

リスト2-3　<htoitest.awk>

```
BEGIN {
        HexPrefix = "^0x"
}

{
        for(i = 1; i <= NF; ++i) {
                if($i ~ HexPrefix) {
                        $i = htoi(substr($i, 3))
                }
        }
        print
}
```

これで、実行時に以下のように-fオプションを複数使用して、この2つのスクリプトを指定すればよい。

```
A>jgawk -f htoitest.awk -f htoi.awk main.c
```

アクションの役割

ルールのアクション部は

いわゆるプログラムに当たる部分

であり、awkの命令を書く。ルールに書かれた命令は、普通のプログラムと同じように頭から順にawkによって実行されていくわけだ。awkの命令さえ知っていれば、アクション部を書くことは特に難しいことはない。

ひとくちにawkの命令と言っても相当な量があるので、ここでは個々の命令の説明は行わない。本書のAPPENDIXにawkの組み込み命令のリファレンスが掲載されているので、目をとおしてみてほしい。

アクションのなかでは、if文やwhile文等の豊富な制御構造も使用することができる。こちらの方もC言語やPASCAL、また最近の強力なBASICを使用したことのあるユーザーであれば、特に注意する点もないのでここでは説明を割愛させていただく。詳しくはAPPENDIXの方を参照していただきたい。それに、こんなところで下手に説明して代わりにAPPENDIXに制御構造の説明がなくなってしまうと、awkのスクリプトを書く際に本書を読もうとしても、命令はAPPENDIX、制御構造は本文の中程なんてことになって、とても使い難いものとなってしまう。これは、

本書の第一号ユーザー

であるはずの私の望むところではないのだ。ということで、アクションについては何も書くことがなくなってしまった。「たまにはこんなこともいいかな？」と思うが、

それではあんまりなので

1つだけ、触れておくことにしよう。

awkのスクリプトを書く際に、アクション部は省略することが可能だ。もちろんアクション部を省略する場合は、パターン部を省略してはならない（理由は簡単、ルールがなくなってしまうからだ）。つまり、パターンだけのルールを書くことができるという意味だ。この場合アクションとして、“{print $0 }”が実行される（ちなみに、print文は、引数を省略すると“print $0”と同じ意味になるので、これは“{print }”と同じ意味になる）。

たとえば、以下のようなスクリプトを実行すると、レコードの中に“<DIR>”という文字があるレコードのみを出力させることができる。

リスト2-4　<act.awk>

```
/<DIR>/
```

<実行結果>

```
A>dir | jgawk -f act.awk
.             <DIR>      92-10-01   23:39
..            <DIR>      92-10-01   23:39
bin           <DIR>      92-12-29   12:39
etc           <DIR>      92-12-26    0:32
fix           <DIR>      93-01-06    1:07
man           <DIR>      92-12-29   12:40
src           <DIR>      92-12-11    0:01
test          <DIR>      92-12-30   13:56
tmp           <DIR>      93-01-31   14:31

A>
```

図2-3　<DIR>レコードの抜き出し <act.awk>

awkの歯顎

これから説明するのはレコードとフィールドの概念である。awkは、あたえられたデータをレコード（多くは行を1レコード）とフィールド（多くはスペースやタブで区切る）に分断して認識したあと処理を始める。

レコードとフィールド

ごはんを咀嚼して、その味をよーく分析して味わうことは大切だ。こうして分析した結果がひょっとして、「麦ごはん」というごはんが、「キーマカレー」というおかずに合う。といった新たな発見を産むかも知れないからだ？ awkも、読み込んだ1レコードのデータをフィールドという単位に

分解・分析して咀嚼する。

awkには、フィールドと、フィールドが集まってできたレコードという概念がある。スクリプトがデータを処理する際にデータはレコード単位で読み込まれ、さらにフィールドへ分解されていく。各アクションでは、特にレコードをフィールドに分解せよといった命令を実行しなくてもこれらのフィールドを処理できるようになっているのだ。

レコード及びフィールドは、指定した文字（正確にはフィールドについては正規表現である）によって区切られる。レコードの区切り文字はRS（レコードセパレータの略）、フィールドの区切り文字はFS（フィールドセパレータの略）に設定する。

これらのシステム変数の規定値は、RSが"¥n"（つまり改行コード）、FSが" "（空白文字）となっている。このため、awkはこれらの値を変更しないかぎり、データファイルから1行分のデータを読み込んではそのデー

タを空白文字で区切られたフィールドに分解してスクリプトを実行する。

1レコード ＝ フィールド＋フィールド＋フィールド…

分解された各フィールドは、$n という変数（n は、始めのフィールドから順に 1,2,3... となる。もちろん即値の代わりに変数を指定することも可能だ）でアクセスすることができる。また$0 は特殊で、レコード全体のイメージが保持されている。

現在処理しているレコードのフィールドの総数は、システム変数 NF（Number of Field）に、現在処理しているレコードの最初のレコードからの通し番号はシステム変数 NR（Number of Record）に入っている。

言葉で説明すると何やら難しくなってしまうが、以下の例を見れば、

一発で理解

できるはずだ。

リスト 2-5　<recfld.awk>

```
{
    printf "$0=¥"%s¥"¥n", $0
    printf "NR=%2d NF=%2d ", NR, NF
    for(i = 1; i <= NF; ++i) {
        printf "$%d=¥"%s¥" ", i, $i
    }
    printf "¥n"
}
```

＜実行結果＞

```
A>dir | jgawk -f recfld.awk
$0=""
NR= 1 NF= 0
$0=" ドライブ B: のボリュームラベルは SD01C        "
NR= 2 NF= 4 $1="ドライブ" $2="B:" $3="のボリュームラベルは"
$4="SD01C"
$0=" ボリュームシリアル番号は 190D-12EE"
NR= 3 NF= 2 $1="ボリュームシリアル番号は" $2="190D-12EE"
$0=" ディレクトリは B:¥HOME¥TAKU¥AWK¥TMP"
NR= 4 NF= 2 $1="ディレクトリは" $2="B:¥HOME¥TAKU¥AWK¥TMP"
$0=""
NR= 5 NF= 0
$0=".              <DIR>      93-01-31   14:31"
NR= 6 NF= 4 $1="." $2="<DIR>" $3="93-01-31" $4="14:31"
$0=".."            <DIR>      93-01-31   14:31"
NR= 7 NF= 4 $1=".." $2="<DIR>" $3="93-01-31" $4="14:31"
$0="recfld   awk        146 93-01-31   15:32"
NR= 8 NF= 5 $1="recfld" $2="awk" $3="146" $4="93-01-31"
$5="15:32"
$0="         3 個          146 バイトのファイルがあります."
NR= 9 NF= 4 $1="3" $2="個" $3="146" $4="バイトのファイルが
あります."
$0="                      66080768 バイトが使用可能です."
NR=10 NF= 2 $1="66080768" $2="バイトが使用可能です."

A>
```

図 2-4　<recfld.awk>の実行

各フィールドを表す$n であるが、なぜこんな名前になっているかご存知だろうか？ 実はこれも UNIX 文化からきているのだ。UNIX の代表的なシェルである sh や csh（最近では ksh や tcsh かな？）のシェルスクリプトのなかでコマンドラインのパラメータを表す変数として$n が使用されており（MS-DOS で言うとバッチの中で使用する%n がこれに当たる）、どうやらこれにあわせて awk で$n が使用されているようなのである。

さて、ここでシステム変数 FS や RS に値を設定する場合に注意しなければならないことがある。

FS には正規表現を指定することが可能だと書いたが、これには条件がある。FS に指定した文字列の長さが 2 文字以上の場合、awk はその文字列

を正規表現として扱うが、1文字の場合はその文字そのものをフィールドセパレータとするのだ。たとえば、FS="."としても"."は正規表現の「任意の1文字」として認識されず、ちゃんと"."(ピリオド)として認識されるわけだ。

FS=" "の場合はさらに特殊になっている。普通に考えればこの場合レコードセパレータはスペースになると考えられるが、実際にはスペースに加えてタブコードがレコードセパレータとなる。

「へー、ということはFS="[¥t]+"と同じになるわけネ」と思った向きはなかなか鋭いが、awkフリークと呼ぶには、

まだまだ超一甘い

FS=" "とFS="[¥t]+"では、その動作に大きな違いがある。FS=" "では、行頭にスペースやタブコードがあった場合、その余白を削除してからフィールドの分解が行われるのに対して、FS="[¥t]+"では、行頭に空白やタブ文字がある場合それもフィールドセパレータとして認識され、最初のフィールドが空文字になるのだ。以下のその違いを示した例を挙げるので、注意して読んでもらいたい。

リスト2-6　" "と"[¥t]+"の違い <fsspc1.awk>

```
BEGIN {
    FS = " "
}
{
    for(i = 1; i <= NF; ++i) {
        printf "%d:[%s] ", i, $i
    }
    printf "¥n"
}
```

＜実行結果＞

```
A>jgawk -f fsspc1.awk
This is test.
1:[This] 2:[is] 3:[test.]
    1st charactor is TAB.
1:[1st] 2:[charactor] 3:[is] 4:[TAB.]
^Z

A>type fsspc2.awk
BEGIN {
    FS = "[ ¥t]+"
}
{
    for(i = 1; i <= NF; ++i) {
        printf "%d:[%s] ", i, $i
    }
    printf "¥n"
}

A>jgawk -f fsspc2.awk
This is test.
1:[This] 2:[is] 3:[test.]
    1st charactor is TAB.
1:[] 2:[1st] 3:[charactor] 4:[is] 5:[TAB.]
^Z

A>
```

図 2-5　<fsspc1.awk>の実行

ちなみにシステム変数 FS の初期値は" "になっている。

「RS には、正規表現が使用できないので難しくないだろう」と、たかを括ってはならない。システム変数 RS の方にも特殊なケースがある。これは、RS=""として RS に空文字を設定した場合であり、この時レコードセパレータは空行となるのである。

よく使用されるケースでは、読み込むデータが CSV (Comma Separated Value) のような 1 行 1 レコードではなく、複数行にわたっているようなものの場合に使用する。

```
ふるいけや
蛙飛び込む
水の音
                          ←この空行がレコードセパレータになる
やせ蛙
頑張れ一茶
ここにあり
```

図2-6　RS=""にすれば…

$n変数は、読むだけではなく書き込むことも可能だ。たとえば以下のような感じに、$1や$0に文字列を代入することができるのだ。

リスト2-7　$1や$0に文字列を代入 <set$.awk>

```
{
    printf "1. $0=¥"%s¥", $1=¥"%s¥"¥n", $0, $1
    $1 = "TEST"
    printf "2. $0=¥"%s¥", $1=¥"%s¥"¥n", $0, $1
    $0 = "I am awk jack."
    printf "3. $0=¥"%s¥", $1=¥"%s¥"¥n", $0, $1
}
```

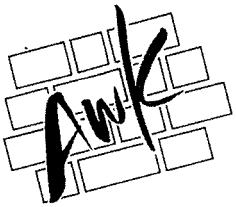
<実行結果>

```
A>jgawk -f set$.awk
Hello, world.
1. $0="Hello, world.", $1="Hello,"
2. $0="TEST world.", $1="TEST"
3. $0="I am awk jack.", $1="I"
^Z

A>
```

図2-7　<set$.awk>の実行

この例を見てもわかるとおり、各フィールドを表す$n（1<=n<=NF）に対して書き込み（もちろん編集操作も含む）を行った場合、即座にレコード全体を表す$0が再構築される。また、逆に$0に対して書き込みを行うと即座に各フィールドを表す$nは再構築される。このように、awkのフィールド分解は

非常に動的に行われるのだ。



awkの胃袋たち

スクリプト中で変数はさまざまな値を格納し、操作する。変数を使いこなすことはプログラミングの作業のなかでも重要度が高いことは言うまでもない。他のプログラム言語に比べて柔軟性に富んでいるawkの変数は、その仕様を把握すれば非常に使いやすい反面、柔軟なだけになかなか理解しにくい面も持っている。

awkの変数

awkの変数は他の真っ当なコンピュータ言語に比べると、少々特殊だ。変数を使用するにあたって特に宣言する必要もないし、数値型や文字型といった変数の型もない。いきなり変数名を書けば使用できるのだ。

このような「書けば使える」といった仕様のため、たとえばawkの命令をミスタイプしても、awkはそれを変数として見てしまい（要するに、awkは自分の知らないトークンはとりあえず変数と見なしてしまうのである）、エラーとも思わず、知らんぷりで実行してしまい変な結果となってしまう場合があるので注意が必要だ。awkはユーザーに対して放任主義を取っているのでユーザー自身がしっかりしなければならないわけだ。「なんだ、このスクリプトは全然うまく動かないじゃないか、クソッ！」なんて吐き捨ててグレみても、awkの責任ではない。

すべて君の責任なのだよ

awkの変数には型がないと書いたが、awkの命令の中には算術命令もあれば、文字列操作の命令もある。どうなっているのだろうか？実はawkの変数には、内部では文字列と数値が入るところが両方用意されていて、

変数が使用されるシチュエーションにあわせて相互の型を変換しているのだ。この暗黙のうちに行なわれる型変換が見えてくるようになれば、awk使いとして中級のレベルに達したと思ってもよいだろう。

最初に変数を使用した時に、変数の既定値は文字列としてアクセスした場合は""（空文字）、数値としてアクセスした場合は0となっている。

"連想配列"はawk独特の機能

awkでは配列変数も使用できる。この配列変数が、普通の配列変数と比べると少々趣の異なるもので、なんと配列の添え字に数値ではなく文字列を指定するようになっているのだ。こうした仕様のため、awkの配列は

「連想配列」という御大層な名前

で呼ばれている。

配列は、配列変数宣言時にその配列の大きさを宣言するのが普通だが、awkの連想配列では他のawkの変数と同じように、使用にあたって何の宣言も必要ない。いきなり使えば連想配列が作成される。しかもメモリが許す限りいくらでも要素を宣言することができるのだ。

連想配列を使用すると、小規模なデータベースを簡単に作成することができる。テカテカになるほど使い込まれた例で申し訳ないが、たとえば英語の月の名前を数字に変換するような場合、以下のように連想配列のテーブルを使用すれば、文字列比較をいっさい行わずに実現することが可能なわけだ。

リスト2-8　月の名前の連想配列 <month.awk>

```
BEGIN {
    Month["January"]   =  1
    Month["Jan."]      =  1
    Month["February"]  =  2
    Month["Feb."]      =  2
    Month["March"]     =  3
```

```
    Month["Mar."]       =  3
    Month["April"]      =  4
    Month["Apr."]       =  4
    Month["May"]        =  5
    Month["June"]       =  6
    Month["Jun."]       =  6
    Month["July"]       =  7
    Month["Jul."]       =  7
    Month["August"]     =  8
    Month["Aug."]       =  8
    Month["September"]  =  9
    Month["Sept."]      =  9
    Month["October"]    = 10
    Month["Oct."]       = 10
    Month["November"]   = 11
    Month["Nov."]       = 11
    Month["December"]   = 12
    Month["Dec."]       = 12
}

{
    print Month[$1]
}
```

＜実行結果＞

```
A>jgawk -f month.awk
Feb.
2
Octorber            ←スペルをミスると、BEGIN で代入されていない添え字で
                    ←配列がアクセスされるので、何も表示されない
October
10
^Z
A>
```

図 2–8　<month.awk>の実行

もちろん、普通の配列のように添え字に数値を使用することも可能だ。これは単に数値を“数字からなる文字列”として添え字にしているのである。

さて一般的に使用される配列は、添え字が数値であるためその配列に入っている要素をなめる場合その配列の要素の数だけループを回せばよいのだが、添え字に文字列が許されている awk の場合、こうした方法は使用できない。

そこで awk では

連想配列をなめるために、

“for(変数 in 連想配列)”という構文が用意されている。これは連想配列で使用されている添え字文字列が順々に変数に代入され、それまでに使用したすべての添え字を変数にするまでループが回る。

リスト 2-9　月の名前の連想配列 <month2.awk>

```
BEGIN {
    Month["January"]   =  1
    Month["Jan."]      =  1
    Month["February"]  =  2
    Month["Feb."]      =  2
    Month["March"]     =  3
    Month["Mar."]      =  3
    Month["April"]     =  4
    Month["Apr."]      =  4
    Month["May"]       =  5
    Month["June"]      =  6
    Month["Jun."]      =  6
    Month["July"]      =  7
    Month["Jul."]      =  7
    Month["August"]    =  8
    Month["Aug."]      =  8
    Month["September"] =  9
    Month["Sept."]     =  9
    Month["October"]   = 10
    Month["Oct."]      = 10
    Month["November"]  = 11
    Month["Nov."]      = 11
    Month["December"]  = 12
```

```
    Month["Dec."]       = 12

    for(name in Month) {
        printf "%s is %s¥n", name,  Month[name]
    }
}
```

<実行例>

```
A>jgawk -f month2.awk
Mar. is 3
Oct. is 10
November is 11
Jul. is 7
Jun. is 6
Sept. is 9
December is 12
Nov. is 11
January is 1
Dec. is 12
February is 2
September is 9
Aug. is 8
August is 8
April is 4
October is 10
Feb. is 2
March is 3
Apr. is 4
Jan. is 1
June is 6
May is 5
July is 7

A>
```

図 2–9　<month2.awk>の実行

上の例を見てもわかるが、実はfor(in)は残念ながら使用した順（要するに時系列にという意味）に添え字が変数に代入されるわけでもなければ、文字コード順に代入されるわけでもない。意味のない順番（実際にはあるロジックによって出現する順番は決まっているのだが）で代入されてしまう。これはバグではなく仕様だ。

潔くあきらめること

また指定した文字列が、配列の添え字文字列として使用されているか否かを調べる方法としてif(in)構文がある。配列の各要素の初期値は、通常の変数と同じように文字列でアクセスした場合は""(空文字)で、数値としてアクセスした場合は0となっている。つまり通常の変数と同じように、ある指定した文字列が添え字として使用されているかを調べるには、

```
if("" == array["script"])
```

などと書くことになる。しかしこうした方法を使用すると、この命令を行った時点で指定した添え字文字列の要素が配列の中に作成されてしまう。そこでawkでは、指定した文字列が添え字として使用されているか否かを調べるためにif(in)という専用の構文が用意されている。

```
if("script" in array)
```

もし文字列が添え字として使用されている場合には、この条件式は値が真となる。

連想配列の実体

連想配列はハッシュテーブルとリンクリストによって実現されている。変数にはハッシュテーブル用に127個ノードがあり、添え字の文字列をawk内部のハッシュ関数にとおすことによって0から126までのキーを返し、このキーの番号のノードにつながっているリンクリストに代入すべきデータをつなげる。

つまり、「連想配列」といった御大層な機能の実体は、

単なるハッシュテーブルだったわけだ。

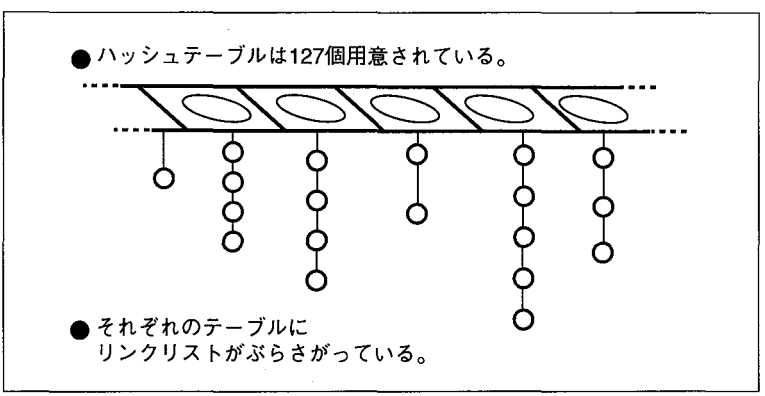


図2-10 ハッシュテーブルとリンクリスト

このハッシュテーブルは、先に通常の変数のところで説明した文字列や数値を保持する部分とは独立した形で保持されているので、1つの変数を通常の変数と配列変数の2つの形で併用することが可能になっている。たとえば、他の言語と同じような添え字に数値しか使用しない配列を作成して、いくつまで添え字を使用したかを保持するような場合、

```
array[arraymax++] = element
```

と書くより、

```
array[array++] = element
```

と書いた方が美しいし（?）変数が少なくなり、メモリをほんの少しだけ節約することができたのだが、最新の2.15.2では変数の扱いが厳密に定義されており、こうした記述は残念ながらできなくなっている。

このような仕様になっているため、メモリが許す限りいくらでも要素が代入できるからといって巨大な連想配列を作ってしまったりすると、配列アクセス時に時間がかかってしまう。特にちょっとawkに慣れてきたユーザーの場合、必要もないのに多次元の連想配列を使用しているケースもあるので、こうした場合は優しく、鼻で笑いながら注意してあげること。

ちなみに変数間の代入は文字列と値の領域がコピーされるだけで、配列のハッシュテーブルやそれに連なっているリンクリストの内容のコピーは行われない。そのため、単に配列変数を等号でつないで代入しようとしても代入は行われない。

さて、awkでは連想配列を使用するたびに、その値を保持するために新たに変数（ノード）を用意する仕組みになっている。awkにはこの動的に確保される変数用メモリを解放する機能も用意されている。delete命令がそれだ。

```
delete array["script"]
```

delete命令は、連想配列arrayの添え字“script”の値を保持しているメモリを解放する。

注意してほしいのは、delete命令でメモリを解放できるのは連想配列の要素として確保された変数（ノード用）メモリについてだけ、という点だ。

ユーザー定義関数

プログラムは何でも関数にしなければ手が震えてくる、といった重度のC言語患者に朗報がある。awkでもユーザー定義関数が使用できるのだ。

awkは、nawkからユーザー定義の関数が使用できるようになっている。もちろん、さらに新しいgawkでも関数定義が可能だ。

関数の定義は、比較的簡単だ。関数名の前に関数宣言を現すトークンの“function”(または“func”)を書く以外は、C言語の関数とほぼ同じような感じだ。「百聞は一見にしかず」とりあえず、ユーザー関数の宣言のサンプルを以下に示す。

```
function 関数名(引数1, 引数2, ...)
{
    awkの命令
    awkの命令
        :
    awkの命令

    return  戻り値
}
```

ユーザー定義関数を呼ぶ側では、引き数に通常の変数や即値はもちろん、連想配列の変数も渡すことができる。関数の手続きは、ここでは2行目と最後の行の“{”と“}”で括られたなかに書けばよい。命令はルールのアクション部で使用できる命令や制御構造がそのまま使用できる。

関数というくらいであるから

戻り値も返すことができる。最後の1つ前の行に書かれているreturn命令がそれを行うための命令だ。return命令を実行すると、その時点で関数の実行を終了し「戻り値」を戻り値として返す。戻り値は、数値でも文字列でも何でもいい。

引き数として連想配列を渡す場合、

通常の変数や即値を渡す場合と動作が若干異なる。ユーザー定義関数側の引き数は、それ専用の変数が用意され、関数を呼んだ側の引き数として指定された内容をその変数にコピーする。まぁ、このようにしておかないと関数を使用する側で即値を指定した場合に問題が起こるのでそうなっているのであろう。問題は、この時にコピーされるのは文字列保持用の領域、数値保持用の領域、それに連想配列用のハッシュテーブルのポインタであるところだ。注意しなければならないのは、連想配列のハッシュテーブルのポインタがコピーされるだけで、実際に要素が格納されているハッシュテーブルや、それにぶら下がっているリスト自体はコピーされない点である。このため通常の変数として使用している部分では、Call by value だが、連想配列の部分に関しては Call by pointer になっているのだ。

こうした仕様のため、連想配列をコピーする時は以下のようなユーザー定義関数を使用すればよい。

```
func arycpy(dst, src,       script)
{
    for(script in src) dst[script] = src[script]
}
```

ちなみに、戻り値の方ではハッシュテーブルのポインタはコピーされないので連想配列を戻り値として返すことはできない。

変数のスコープ管理

awk のスクリプトのなかで使用している変数は、一部の例外を除いてすべて全域でアクセスすることができる。たとえばスクリプトのなかに 2 つのルールがあって、1 つ目のルールで設定した変数を、2 つ目のルールで何の制限もなく読み書きすることができるのだ。つまり、変数のスコープ管理といった

高尚なものはawkには存在しない

（この後説明する唯一の例外を除いて）

この仕様の例外が、前節で少し説明したユーザー定義関数の引数に指定した変数である。よく考えてみれば、この引き数にさえスコープ管理がないとすると、引き数を指定する必要がないのだから（つまりパラメータという形をとらなくても、ある決まった変数に指定したい値を設定して、ユーザー定義関数を呼べばよいわけだ）これはawkの言語仕様を知らなくても、容易に想像がつくはずだ。

ユーザー定義関数の引数として宣言されている変数は、その関数内でのみ有効というスコープを持っている。このためユーザー定義関数のなかでローカルな変数を使用したい場合、引数として使用しなくても引数として宣言して関数内ローカルのスコープを持つ変数を用意するといった手法を用いる。

つまり、引数を2つ必要とするユーザー定義関数でローカル変数を用意したい場合は、関数宣言で本当に引数として使用する2つの変数に続けてローカル変数として使用したい変数を引数として記述するのだ。（参照：3章 変数を自在に）

リスト2-10 ユーザー定義関数でのローカル変数定義

```
# このユーザー関数のpromptとdefaultの2つが実際の引数で、その後ろ
# に書かれているyornとansは、関数内で一時的に使用するためにロー
# カル変数として用意したもの。実際には以下のようにして使用する。
#
#       ret = Sure("頭は大丈夫ですか", "N")
#
# どこまでが引数でどこからがローカル変数を実現するための引数なの
# か区別するために、通常このようにそれらの分かれ目にタブやスペー
# スで区切りを付けて表記するのが仁義になっている。

func Sure(prompt, default,      yorn, ans) {
        default = toupper(substr(default, 1, 1))
        if(     default == "Y") yorn = "([Y]orN)"
        else if(default == "N") yorn = "(Yor[N])"
        else                    yorn = "(YorN)"
```

```
        printf "%s%s? ", prompt, yorn
        do {
                getline ans < "/dev/stdin"
                if("" == ans) ans = default
        } while(ans !~ /^[YyNn]/)
        return ans
}
```

awkではユーザー定義関数の宣言と、その関数を使用する側で引数の数が合わなくてもエラーとならないのでこうした書き方をしても問題ないわけだ。ちなみに関数を呼ぶ側で値の与えられなかった引数（つまり、ローカル変数として宣言した引数）は、未使用の変数と同じように文字列としては""、数値としては0となっている。

さて、こうした引数の数が合わなくてもエラーとならないというawk独特の仕様は、見方によっては

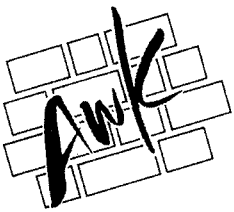
非常に危険だ。

ユーザー定義関数を利用する場合は、その引数の数に十分気をつける必要がある。

awkの舞台裏を探る!

—プログラマーズTIPS—

さぁ、いよいよ後半に突入する。本章では、awkでスクリプトを書くときに、知っていると便利な小技や豆知識をまとめて紹介しようと思う。**プログラマーズTIPS**だ。言語仕様を学んで、プログラムが一通り書けるようになったからといって、「もうawkは理解した」などと思ってもらっては困る。そう、ようやくスタートラインにつけた段階なのだ。これから説明する、**マニュアルに記載されていない**ような細かな仕様や癖を覚えてこそ、真のawkスクリプターとなれるのだ。キッシュ・イーター(古い!)にならないためにも、本章を読破することをお薦めする。



文字列を自在に

問題: awkでは、文字列を出力するのにいくつかの方法が用意されていますね。それでは質問です。そのなかで最も速い文字列出力は、何のコマンドでしょう？

結論: こんなことも調べずに安穏とawkってる向きは…

この節を読むように!

文字列をつなぐ

文字列をつなぐには、2つの文字列をただ空白で区切って並べて書けばよい。以下の例では、変数cの内容は"ABCDEFGHI"になる。

```
a = "ABC"
b = "DEF"
c = a b "GHI"
```

初心者、ことBASICやC等の他の言語を知っている初心者にとってawkで2つの文字列をつなぐ方法は意外と知られていない。これは、2つの文字列をつなぐための演算子といったものが存在しないというawkの仕様が原因だろう。

そこで彼らは、おなじみのsprintf()関数を使用して無理やり文字列の連結を行ってしまう。もちろんsprintf()関数を使用するより空白で区切って並べた方が、若干ながらスピードも速い。

文字列を出力する

awkで文字列を出力するには、printとprintfの2つのコマンドが用意されている。厳密には、printfは書式制御ができる点でprintとは異なるが、単なる文字列定数や、特に書式を指定しないで文字列変数を出力したい場合、どちらのコマンドを使用すればよいのだろう。

調べてみると意外!

なことに、printコマンドを使用するよりprintfコマンドで出力した方が、若干速度が速いことがわかった。これはMS-DOS版のjgawk.exeをPC-9801RA21（Cx486DLC,NDPなし）の筆者の環境で実行した結果だ。マシンやawkの処理系によって異なる可能性も考えられるので、あなたの環境を、次に示すスクリプトで一度実験してみるといいかも知れない。

リスト3-1　<chkpp.awk>

```
BEGIN {
    i = 0
    a = "1234567890"
    b = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    f = "/dev/stderr"

    print "Print & Printf Speed Check"              >> f
    print "-------------------------"               >> f

    printf "1.1. print a                : "         >> f
    srand()
    for (i = 0; i < 30000; ++i) print a
    print -srand()+srand() "sec"                    >> f

    printf "1.2. printf ¥"%s¥¥n¥", a       : "      >> f
    srand()
    for (i = 0; i < 30000; ++i) printf "%s¥n", a
    print -srand()+srand() "sec"                    >> f

    printf "1.3. printf a ¥"¥¥n¥"          : "      >> f
```

```
    srand()
    for (i = 0; i < 30000; ++i) printf a "¥n"
    print -srand()+srand() "sec"                    >> f

    printf "2.1. print a, b             : "         >> f
    srand()
    for (i = 0; i < 30000; ++i) print a, b
    print -srand()+srand() "sec"                    >> f

    printf "2.2. print a ¥" ¥" b          : "       >> f
    srand()
    for (i = 0; i < 30000; ++i) print a " " b
    print -srand()+srand() "sec"                    >> f

    printf "2.3. printf ¥"%%s %%s¥¥n¥", a, b : "    >> f
    srand()
    for (i = 0; i < 30000; ++i) printf "%s %s¥n", a, b
    print -srand()+srand() "sec"                    >> f

    printf "2.4. printf a ¥" ¥" b ¥"¥¥n¥"    : "    >> f
    srand()
    for (i = 0; i < 30000; ++i) printf a " " b "¥n"
    print -srand()+srand() "sec"                    >> f
}
```

<実行結果>

```
A>jgawk -f chkpp.awk > nul
Print & Printf Speed Check
--------------------------
1.1. print a                : 29sec
1.2. printf "%s¥n", a       : 27sec
1.3. printf a "¥n"          : 30sec
2.1. print a, b             : 45sec
2.2. print a " " b          : 34sec
2.3. printf "%s %s¥n", a, b : 31sec
2.4. printf a " " b "¥n"    : 38sec

A>
```

図 3–1　<chkpp.awk>の実行

1.1 から 1.3 の変数 a のみを出力するだけの場合は、1.2 が若干速いよう

である。2.1から2.4の変数aとbを出力する場合は、2.1のprint命令を使った方法は非常に遅い。Cプログラマは、printf関数というと高価な関数でスピードが遅いのでなるべく使用しないようにする向きも多いが、

awkの場合はそうでもないようだ。

文字列の長さを調べる

文字列の長さを調べるlength()関数は、

C言語のstrlen()のように高価

な関数ではない。本章「変数を明示的に制御する」の項でも説明しているが、awkの変数は、文字列とともにその文字列の長さをも保持しているので（直接的には見えないが）length()関数は、その値を返すだけなのだ。

ただしjlength()関数については、内部で文字列をループを回して算出しているので、C言語のstrlen()と同じように高価な処理になっている。

文字列比較に大文字小文字の区別をなくす

文字列を比較したり検索したりする場合、大文字（全角という意味ではない）と小文字を区別しない方が都合がいい場合がある。こうした場合、単なる文字列比較であればtoupper()関数やtolower()関数を使用して、大文字または小文字に

強制的に文字列を変換

して比較を行えばよい。

しかしこの方法では、'~'照合演算子を使用して正規表現の照合を行うような場合にうまくいかないケースが出てくる（たとえば/[I–i]/といった正

規表現を使用した場合など)。こういった場合はIGNORECASEシステム変数を使用すればよい。IGNORECASEシステム変数を0以外にすると、すべての正規表現の照合において大文字小文字の区別をしなくなる。

ここで注意しなければならないのは、FSシステム変数に設定したフィールドセパレータも大文字小文字を無視する点だ。また

念のためにしつこく言っておく

と、IGNORECASEシステム変数を0以外に設定しても、単純な文字列比較の場合は大文字小文字を区別して比較される。

このIGNORECASEシステム変数はgawk特有のものだ。

文字列の長さ

awkのマニュアルには、awkで扱う文字列の長さに制限がないと書かれている。まぁ、プログラムの構造的にはという意味なのだろうが、

それにしたっていい加減すぎる。

そこで変数に文字列を連結していき、どのくらいの長さで実行エラーとなるか調べてみることにした。以下に示すのが使用したスクリプトと、その結果だ。

スクリプトは最初に1KByteの文字列を作成し、これを無限ループのなかで用意した変数に連結していくものだ。

リスト3-2　<strlen.awk>

```
BEGIN {
    for(i = 0; i < 1024; ++i) {
        kilo = kilo " "
    }
    i = 0
    while(1) {
        printf "%dK ", ++i
```

```
            s = s kilo
        }
    }
```

<実行結果>

```
A>jgawk -f strlen.awk          ←通常の jgawk
1K 2K 3K 4K 5K 6K 7K 8K 9K 10K 11K 12K 13K 14K 15K 16K 17K
18K 19K 20K 21K 22K 23K 24K 25K 26K 27K 28K 29K 30K 31K 32K
33K 34K 35K 36K 37K 38K 39K 40K 41K 42K 43K 44K 45K 46K 47K
48K 49K 50K 51K 52K 53K 54K 55K 56K 57K 58K 59K 60K 61K 62K
63K 64K 65K 66K 67K 68K 69K 70K 71K 72K 73K 74K 75K 76K 77K
78K 79K 80K 81K 82K 83K 84K 85K 86K 87K 88K 89K 90K 91K 92K
93K 94K 95K 96K 97K 98K 99K 100K 101K 102K 103K 104K 105K
106K 107K 108K 109K 110K 111K 112K 113K 114K 115K 116K 117K
118K 119K 120K 121K
jgawk.exe: fatal error: tree_eval: var: can't allocate
memory source line number 3, file 'ttt'
```

図 3–2　<strlen.awk>の実行例（標準版）

```
A>jgawk87 -f strlen.awk        ← NDP 版の jgawk
1K 2K 3K 4K 5K 6K 7K 8K 9K 10K 11K 12K 13K 14K 15K 16K 17K
18K 19K 20K 21K 22K 23K 24K 25K 26K 27K 28K 29K 30K 31K 32K
33K 34K 35K 36K 37K 38K 39K 40K 41K 42K 43K 44K 45K 46K 47K
48K 49K 50K 51K 52K 53K 54K 55K 56K 57K 58K 59K 60K 61K 62K
63K 64K 65K 66K 67K 68K 69K 70K 71K 72K 73K 74K 75K 76K 77K
78K 79K 80K 81K 82K 83K 84K 85K 86K 87K 88K 89K 90K 91K 92K
93K 94K 95K 96K 97K 98K 99K 100K 101K 102K 103K 104K 105K
106K 107K 108K 109K 110K 111K 112K 113K 114K 115K 116K 117K
118K 119K 120K
jgawk87.exe: fatal error: tree_eval: var: can't allocate
memory source line number 3, file 'ttt'
```

図 3–3　<strlen.awk>の実行例（NDP 版）

```
A>jgawk32 -f strlen.awk      ← GO32 版の jgawk
jgawk32.exe: fatal error: tree_eval: var: can't allocate
memory source line number 3, file 'ttt'
1K 2K 3K 4K 5K 6K 7K 8K 9K 10K 11K 12K 13K 14K 15K 16K 17K
18K 19K 20K 21K 22K 23K 24K 25K 26K 27K 28K 29K 30K 31K 32K
```

図 3-4　<strlen.awk>の実行例（GO32 版）

実行結果については、環境によって必ずしもこれと同じになるとは限らないが、筆者の環境では、GO32 を使用していないものでは大体 120KByte 強までの文字列を作成することができた。

目を疑ったのが、GO32 を使用した方の結果である。なんと、32KByte の文字列を作成した時点でメモリが確保できず、実行時エラーとなってしまった。

がぁーん

固定長のデータを読むには

CSV や SYLK 等のように、フィールドとフィールドのわかれ目が特定の文字に決められている場合、FS 変数を変更することによって、awk でも簡単にデータとして扱うことができるようになる。しかし、残念ながら世の中にはフィールドとフィールドのわかれ目が文字になっていないデータが存在する。このようなデータの多くは、各フィールドの長さが固定になっている、

固定長フィールドデータと呼ばれるものだ。

固定長フィールドのデータを awk で扱うには、どのようにするのか？答えは簡単、substr 関数を使用して、自分で分解するしかないのだ。
固定長データの例を挙げよう。たとえば DIR の出力は、ファイル 8、拡張子 3、ファイルサイズ 10 などと区切られている。これを awk で各フィー

ルドに分解するには、以下のようにsubstr関数を使用する。

リスト3-3　<dir.awk>

```
/^[ ¥t.]/ || /^$/ {
    next
}
{
    file = substr($0,  1,  8)
    ext  = substr($0, 10,  3)
    size = substr($0, 14, 10)
    date = substr($0, 24,  8)
    time = substr($0, 35,  5)
}
{
    sub(/ *$/, "", file)
    sub(/ *$/, "", ext)
    print file "." ext
}
```

<実行結果>

```
A>dir | awk -f dir.awk
test.
tmp.
act.awk
awk.1
awk1_spl.80
awk256_2.80
awk256_3.bak
awk256_3.80
awkpp.bak

A>
```

図3-5　<dir.awk>の実行

ちなみに新しい gawk(2.13 以降) では、固定長データのアクセスのために FIELDWIDTHS 変数が用意されており、

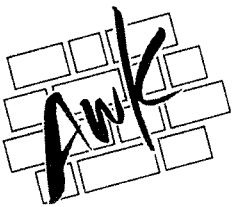
こうした処理をもっとスマートに

行うことができるようになっている。

FIELDWIDTHS 変数に各フィールドの長さを数値を空白文字で区切って書くことによって、$n に指定されたフィールド長のデータが格納されるのだ。先の substr 関数を使用したスクリプトを、この FIELDWIDTHS 変数を使用して書き直したものが以下のリストだ。

リスト 3-4　FIELDWIDTH 変数を使って

```
BEGIN {
    FIELDWIDTHS = "9 4 10 9 8"
}
/^[ ¥t.]/ || /^$/ {
    next
}
{
    sub(/ *$/, "", $1)
    sub(/ *$/, "", $2)
    print $1 "." $2
}
```



変数を自在に

問題: awkの配列といえば連想配列ですね。それではこの連想配列を"for in"ループを使用してなめた時にその出現順が順不同なのは、何故でしょうか?

結論: こんなことも知らずに安穏とawkってる向きは…

この節を読むように!

変数を明示的に制御する

awkの変数には、他の言語にあるような整数型や文字型などといった型が存在しない。どの変数でも数値や文字列を自由に代入することができる。しかし、awkの内部でも変数の型が存在しないわけではない。

awkでは、さまざまな値や変数、トークン等をノード(node)という1つの型で表している。このノードで変数を表す場合、その変数の数値を保持するnumber、文字列を保持するsp、文字列の長さを保持するslen、それにどちらのタイプの領域の値が有効であるかを保持するためのflagsというメンバが用意されいてる(このあたりの宣言はawk.hに以下のように書かれている)。

リスト3-5　<awk.h>より

```
    unsigned char flags;
#           define  MEM 0x7
#           define  MALLOC  1   /* can be free'd */
#           define  TEMP    2   /* should be free'd *
```

```
#           define  PERM    4   /* can't be free'd *
#           define  VAL 0x18
#           define  NUM 8   /* numeric value is valid */
#           define  STR 16  /* string value is valid */
#           define  NUMERIC 32  /* entire field is numeric */
```

数値を変数に代入した場合、その値だけがnumberに保持される。その変数を文字列に変換する必要ができた時点（たとえば、print文で内容を出力する場合など）で文字列に変換され、spに保持される（この時にslenも設定される）。この

数値から文字列への変換

は、システム変数OFMTに設定されている書式指定文字列にしたがって行われる。

OFMTは、初期値が"%.6g"となっているので、6桁を越える数値の場合は以下のように変換されてしまう。

リスト3-6　<cast.awk>

```
BEGIN {
    val = 1234567890
    print "1) val=", val, "len=", length(val)
    OFMT = "%.10f"
    print "2) val=", val, "len=", length(val)
}
```

<実行結果>

```
A>jgawk -f cast.awk
1) val= 1.23457e+009 len= 12
2) val= 1.23457e+009 len= 12.0000000000
```

図3-6　<cast.awk>の実行例

このテストで興味深いのは、OFMTを"%.10f"に変更したにもかかわらず、その後に出力している

valの値が変わっていない

ことだ。length()関数の返り値は、キチンと"%.10f"にしたがった形で出力されているところを見ると、OFMTの設定はうまくいっている。

これは、はじめのprint文を実行した際に、変数valの文字列型の値として数値123456を"%.6g"で変換した"1.23457e+009"という文字列が設定され、2回目のprintでvalを文字列として出力する際には"1.23457e+009"という文字列がすでに設定されているので、その値が出力されているのだ。

もし、valを期待どおりに、新たにOFMTに設定した書式にしたがって表示したい場合は以下のようにすればよい。

リスト3-7　<cast2.awk>

```
BEGIN {
    val = 1234567890
    print "1) val=", val, "len=", length(val)
    OFMT = "%.10f"
    val += 0
    print "2) val=", val, "len=", length(val)
}
```

<実行結果>

```
A>jgawk -f cast2.awk
1) val= 1.23457e+009 len= 12
2) val= 1234567890.0000000000 len= 21.0000000000
```

図3-7　<cast2.awk>の実行

これは、val += 0によってvalに新しく値が設定されたことになり、それまで保持していた"1.23457e+009"という文字列が無効となり、2回目

のprint文実行時に、再度数値から文字列への変換がOFMTシステム変数にしたがって行われた結果、期待どおりの結果となったわけだ。

これと同じように、awk内部で

保持している変数の値を無効にすることも

val = val ""とすることで可能である。しかし、主に数値を扱う変数に対してこの処理を行うことは、非常に危険だ。

リスト3-8　<cast3.awk>

```
BEGIN {
        val = 1234567890
        print "val=", val
        print "1) val - 1234500000 =", val - 1234500000
        val = val ""
        print "2) val - 1234500000 =", val - 1234500000
}
```

<実行結果>

```
A>jgawk -f cast2
val= 1.23457e+009
1) val - 1234500000 = 67890
2) val - 1234500000 = 70000
```

図3-8　<cast2.awk>の実行

もうわかっただろう

val = val ""を実行した時点で、数値として保持していた1234567890という値が無効となり、変数valは"1.23457e+009"という文字列のみが保持されるだけとなり、次の行のprint文でvalを数値として扱う必要から、この文字列を数値に変換し、valが1234570000という値になり、このよう

な結果となったわけだ。

ちなみに、gawkのVer.2.13以降では、awkの内部で行われる文字列型から数値型への型変換は、OFMTとは別に、それ専用のCONVFMTという新しいシステム変数が用意されているので注意が必要だ。OFMTを型変換の書式として設定しているスクリプトでは、CONVFMTにもその文字列を設定しておかないと、期待どおりにスクリプトが実行できなくなる。

変数が使用されているか否か調べる

awkでは変数を使用するにあたって特に宣言を必要としない。このため、ある変数がすでに使用されているか否かを調べる必要が出てくる場合がある。しかしawkには、それを調べる方法は、

残念ながら用意されていない。

awkで変数を使用する場合、変数の初期値として、文字列としてアクセスした場合""(空文字)、値としてアクセスした場合0が設定されるようになっている。そこでこれを利用して、変数が使用されているか否かは、以下のように検査するのが一般的となっている。

```
if ("" == val) {
    # 変数valは、まだ使用されていない。
}
```

この方法では変数の値が空文字の場合、変数は使用されていないということになってしまうが、これは

もう判別のしようがない

のであきらめる。

さて、これが連想配列に指定した添え字の変数が使用されているか否か

になると、確実に調べることができる。これには in 演算子を使用して以下のようにすればよい。

リスト 3-9　<in_in.awk>

```
BEGIN {
    rgb["blue"]  = 1
    rgb["red"]   = 2
    rgb["green"] = 4

    if("yellow" in rgb) {
        printf "yellow is exist.¥n"
    } else {
        printf "yellow is nothing.¥n"
    }

    if("blue" in rgb) {
        printf "blue is exist.¥n"
    } else {
        printf "blue is nothing.¥n"
    }
}
```

<実行結果>

```
A>jgawk -f if_in.awk
yellow is nothing.
blue is exist.
```

図 3-9　<if_in.awk>の実行

in 演算子は、"<添え字>in <配列>"といった形で使用し、<配列>の中で<添え字>が添え字としてすでに使用されていれば真、そうでなければ偽を返す。

テーブル用配列を初期化する

たとえば、指定した数値を16進数の文字列に変換するルーチンを作成するような場合（実際にはsprintf()関数を使用するだろうが）、配列変数に変換表を作成し、それを参照しながら作業を行うのが一般的だろう。この時、以下のようにBEGINの中で、

徒然なるままに

テーブルの初期化を行うのは美しくない。

```
_ItohTbl[0] = "0"
_ItohTbl[1] = "1"
_ItohTbl[2] = "2"
_ItohTbl[3] = "3"
    :
    :
```

こういう場合は以下のようにsplit()関数を使用すれば、スッキリと書くことができる。

```
split("1 2 3 4 5 6 7 8 9 A B C D E F", _ItohTbl, " ")
_ItohTbl[0] = "0"
```

あー、すっきり

配列の中をさらう

他の言語であれば、配列の添え字は数値であり、その数値も1、2、3といった風に単純にリニアに増えていくことになっているので、添え字の最大の数（もしかすると、最低の数も必要化もしれないが）さえ覚えておけば、その配列の全内容をループを回して簡単に参照することができる。

ところが

awkの配列は連想配列

であり、配列の添え字として文字列が使用できるようになっているのでそうは簡単に話が進まない。このため、awkでは連想配列の全内容を参照するために、"for(<添え字>in <配列>)"といった構文が用意されている。

この構文は、現在までに<配列>の添え字として使用された文字列が、<添え字>に指定された変数に設定され、全部の添え字を<添え字>に設定するまでループが回る。その動作を試すために、以下のようなスクリプトを用意し、実行してみた。

リスト3-10 <for_in.awk>

```
BEGIN {
    rgb["blue"]  = 1
    rgb["red"]   = 2
    rgb["green"] = 4
    for(s in rgb) {
        print "rgb[" s "] = " rgb[s]
    }

    split("1 2 3 4 5 6 7 8 9 0", array, " ")
    for(s in array) {
        print "array[" s "] = " array[s]
    }
}
```

<実行結果>

```
A>jgawk -f for_in.awk
rgb[green] = 4
rgb[blue] = 1
rgb[red] = 2
array[4] = 4
array[8] = 8
array[1] = 1
array[5] = 5
array[9] = 9
array[2] = 2
array[6] = 6
array[10] = 0
array[3] = 3
array[7] = 7
```

図3-10 <for_in.awk>の実行

ここで注目すべき点は、この構文が必ずしも添え字を使った順にループが回らないという点だ。この点についてはマニュアルでも言及されている。gawkでは、連想配列は

添え字をキーとしたハッシュテーブル

を使用して配列の内容を保持している。このあたりは、array.cを見ればわかる。

"for(<添え字>in <配列>)"は、配列のハッシュテーブルをなめて添え字を取り出すが、ハッシュテーブルは、ハッシュ関数（array.cのなかのhash_calc()関数を参照のこと）によって

順序がバラバラになっている

（バラバラになっていなければハッシュにならないのだが）ので、添え字が使用した順番に現れないのである。

多次元配列を利用する

awkの連想配列は、前章の連想配列のところで説明したように、要素数が多くなると各要素に対するアクセスが遅くなるので、あまり大きな配列を作成するより、できるなら複数の配列に分けた方が効率がよい。つまりawkでは、通常のプログラムのように複数の配列を多次元配列にまとめてプログラムをスッキリさせるといったことは

行わないほうが無難である。

とは言っても、効率を無視しても多次元配列を使用したい場合や、それほど大きな配列にはならない多次元のテーブルを作成したい場合もある。こんな場合のために、awkでは（模倣ではあるが）多次元の配列も用意されている。多次元配列を使用するには、以下のようにして、添え字をカンマで区切って書くだけだ。

```
array[script1, script2, ... scriptN]
```

もちろん普通の連想配列と同じように、添え字として文字列を使用することもできるし使用するにあたって特に宣言することもいらない。それどころか、異なった次元数の配列を1つの配列変数で扱うことすら可能だ。

こうした柔軟な仕様は、多次元配列の実装方法を知れば納得できる。awkの多次元配列は、

実は見かけ上だけ

のことで、実際には1次元の連想配列になっているのだ。多次元配列として複数の添え字が指定された場合、awkはそれぞれの添え字文字列を、システム変数SUBSEPで指定された文字でつなぎ、1つの文字列に再構成する。そしてその文字列を添え字文字列とし、通常の連想配列の添え字として使用するわけだ。

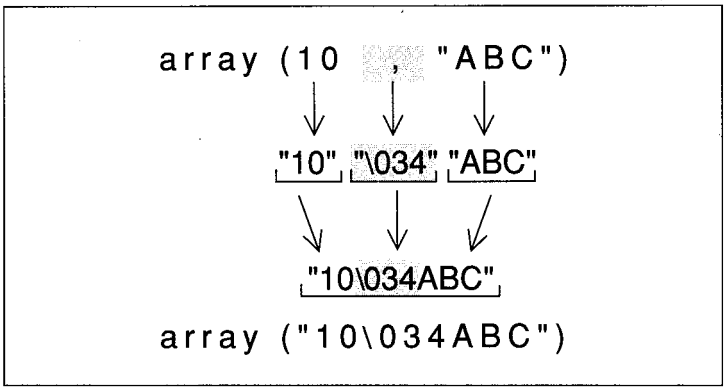


図3-11 連想配列の内部構造

システム変数SUBSEPには、文字コードで、"¥034"という、通常の文字列と指定できない文字が指定されているので、添え字とその区切り文字の区別がつかなくなることがないわけだ。

この仕組みを確認するには、システム変数に通常の文字列として使用できる文字を指定すればよい。たとえば以下のような感じだ。

リスト3-11 <mltary.awk>

```
BEGIN{
    SUBSEP = ":"

    array[1, "ABC"] = "当たり!"

    print "1." array[1, "ABC"]
    print "2." array["1", "ABC"]
    print "3." array[1 ":" "ABC"]
    print "4." array["1" ":" "ABC"]
    print "5." array["1:ABC"]
}
```

＜実行結果＞

```
A>jgawk -f mltary.awk
1.当たり!
2.当たり!
3.当たり!
4.当たり!
5.当たり!

A>
```

図 3-12 <mltary.awk>の実行

多次元配列を使用した場合、in 演算子を使用する時にはちょっとしたコツがいる。in 演算子を使用して、指定した文字列が添え字として使用されているか否かを調べる場合、以下のように添え字をカンマで区切り、さらに括弧で括って in 演算子の左辺として指定すればよい。

```
(script1, script2, ... scriptN) in array
```

for(in) 構文の場合、残念ながらこのような記述はできない。そこで多次元配列で for(in) 構文を使用するには、以下のように split 命令を使用して取得した添え字文字列をシステム変数 SUBSEP で分解する。

```
for(script in array) {
    split(script, list, SUBSEP)
        :
        :
}
```

関数の中でローカル変数を用意する

awkでは、

関数宣言も可能

となっている。この関数を使用する際に問題になってくるのが変数のスコープ管理だ。awkでは、1つの例外を除いて基本的に変数にはスコープがなく、すべてグローバル変数となっている。

この1つの例外が、関数宣言時に指定する引き数の変数だ。awkの関数の引き数は、普通、

Call By Value(値渡し)

で、変数はその関数内のみ有効なスコープを持つ。

さて、ここで実際にその関数を呼ぶ際に指定する引き数よりも多くパラメータを指定しておくとどうなるか。結果的には何の問題もない。引き数の渡されなかったパラメータは、新しいローカル変数として用意される。

そこでawkで関数を作成する場合は、ローカル変数として関数の引き数宣言を利用することが一般的になっている。

```
func name(param1, param2,        local1, local2, ...) { }
```

このあたりまではawkプログラマなら常識だ。それでは配列を引き数として宣言した場合はどうなるのだろう(これも、知ってる人は少なくないか)? これは以下のような簡単なスクリプトで確かめることができる。

リスト3-12 <aryscp.awk>

```
BEGIN {
    a[1] = 456
    print "in BEGIN(1) a[1]=" a[1]
    aryscp(a)
```

```
    print "in BEGIN(2) a[1]=" a[1]

    b[1] = 456
    print "in BEGIN(3) b[1]=" b[1]
    aryscp(b)
    print "in BEGIN(4) b[1]=" b[1]
}

func aryscp(a)
{
    print "in aryscp a[1]=" a[1]
    a[1] = 123
    print "in aryscp a[1]=" a[1]
}
```

<実行結果>

```
A>jgawk -f aryscp.awk
in BEGIN(1) a[1]=456
in aryscp a[1]=456
in aryscp a[1]=123
in BEGIN(2) a[1]=123
in BEGIN(3) b[1]=456
in aryscp a[1]=456
in aryscp a[1]=123
in BEGIN(4) b[1]=123
```

図 3-13 <aryscp.awk>の実行

この実験からもわかるように、引き数に配列を使用した場合はその配列の

各要素の値は素どおし

になっている。これはちょうどC言語で配列をポインタで渡した状態と似ている。配列変数は、その変数のノードに配列要素のポインタリストのポインターハッシュテーブルを持っている。配列変数を関数の引き数として引き渡すとき新しいノードが作成され、そこにこれらの

ポインタがコピーされるだけ

なので、結果的にコールバイポインターになり、配列の要素を変更すると関数を呼んだ側の変数の内容も変わってしまうわけだ。

システム変数の初期値を変更する

たとえば、CSVのファイルを扱う場合は、FSを","にする。いつもCSVを専門に扱う向きは、FSの規定値を","にしておけば、いちいち-FオプションやBEGINアクションでFSの値を変更せずにすむので、

極楽だ!

これはawkのソースをコンパイルできる環境があれば簡単に変更することができる。jgawkの場合は、main.cの中のinit_vars()関数の中で、これらのシステム変数の初期化が行われている。たとえばFSの規定値は、以下のような形で宣言されている。

```
FS_node = spc_var("FS", make_string(" ", 1));
```

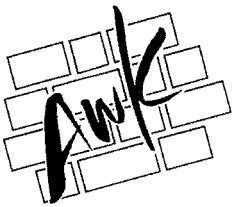
たとえば、これを先に書いたとおり、","に変更するには、以下のように変更しリビルドすればよい。

```
FS_node = spc_var("FS", make_string(",", 1));
```

この他にも、この関数の中では、ほとんどのシステム変数の初期化が行われているので、適当に変更し

専用のjgawkを作ってみる

のも面白いかもしれない。



awkをより速く

問題: 実行するのに11秒かかっていたスクリプトを、ある操作を行うことによって7秒に短縮できた。さて、何を行ったのでしょう？ ただしマシンはPC-9801RA21を使用したものとする。

結論: これくらいのことが想像できない向きは…

この節を読むように!

NDPノススメ

awkでは、数値演算の多くを倍精度浮動小数点型（C言語でいうところのdouble型）で行う。数値演算コ・プロセッサ（以下NDPと略す）が実装されていないパソコン等のコンピュータでは、この演算をCPUを使用して、何バイトかのコードでエミュレーションしている。

NDPを実装できるコンピュータでは、NDPを実装することによってawkの数値処理を高速化することができる。以下に簡単なベンチのためのスクリプトを用意して実行した結果を表示する。NDP版では

forループが高速

になっていることがわかる。逆に文字列の処理は特に速くなっていない。

リスト3-13 <cpuspeed.awk>

```
BEGIN {
    i = 0
    a = "1234567890"
    b = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

```
    f = "/dev/stderr"

    print "CPU Speed Check"          >> f
    print "---------------"          >> f

    printf "1. for   : "             >> f
    srand()
    for (i = 0; i < 30000; ++i) ;
    print -srand()+srand() "sec"     >> f

    printf "2. print : "             >> f
    srand()
    for (i = 0; i < 30000; ++i) print a
    print -srand()+srand() "sec"     >> f

    printf "3. gsub  : "             >> f
    srand()
    for (i = 0; i < 30000; ++i) {
        a = "0A1BC2DEF3GHIJ4KLMNO5PQRSTUV6WXYZ7"
        gsub(/[0-9]/, "#", a)
    }
    print -srand()+srand() "sec"     >> f
}
```

<実行結果>

```
A>jgawk -f cpuspeed.awk > nul        ←通常の jgawk
CPU Speed Check
---------------
1. for   : 11sec
2. print : 28sec
3. gsub  : 70sec

A>jgawk87 -f cpuspeed.awk > nul      ← NDP 版 jgawk
CPU Speed Check
---------------
1. for   : 7sec
2. print : 25sec
3. gsub  : 65sec
A>
```

図 3-14 <cpuspeed.awk>の実行

最近のプログラムでは、NDPがあれば浮動小数点演算をNDPにさせるようになっているものも多いので、ここは

NDPをおごってやることをオススメする。

80386以上のCPUでawkを使用する

CPUが80386以上のCPUでは、DOSエクステンダを使用したawkが使用できる。jgawkは、2.11.1＋3.0からGO32を利用したjgawkの実行ファイルが添付されている。このバージョンでは、ほとんどの部分が80386のプロテクトモードで実行されるので広大なプロテクトメモリーをダイレクトに利用できる。さらにGO32には

仮想記憶機能

がついており、メモリをページングすることで物理メモリ以上のメモリを使用することができる。このバージョンを使用すれば、相当大規模な連想配列も余裕でこなすことが可能だ。また、処理スピードについてもNDPさえあれば、おおむね通常のjgawkに比べて高速である。

一見いいことづくめのGO32版jgawk

であるが、盲目的にこれを使用してはならない。ここで、GO32版jgawkを利用するにあたって注意しなければならない点を2つ説明しておこう。

第一点は、NDPが実装されていない場合、実行スピードが、非常に遅くなってしまう点だ。簡単なベンチをしてみた結果を以下に示すが、これを見てもわかるように通常のjgawkよりもだいぶ遅い結果となってしまう。

```
A>jgawk -f cpuspeed.awk > nul          ←通常の jgawk
CPU Speed Check
---------------
1. for   : 11sec
2. print : 28sec
3. gsub  : 70sec
```

図 3-15 <cpuspeed.awk>の実行例（標準版）

```
A>set GO32=emu a:¥usr¥lib¥emu387

A>jgawk32 -f cpuspeed.awk > nul        ← GO32 版 jgawk(NDP なし)
CPU Speed Check
---------------
1. for   : 36sec
2. print : 86sec
3. gsub  : 101sec
```

図 3-16 <cpuspeed.awk>の実行例（GO32 版 jgawk NDP なし）

第 2 点は、print 文などの MS-DOS にリクエストを発行するような処理が多い場合も、スピードが遅くなってしまう点だ。これも以下に簡単なベンチを以下に示す。NDP がない場合と同じように、通常の jgawk よりも遅くなってしまうケースがある。

```
A>set GO32=

A>jgawk32 -f cpuspeed.awk > nul        ← GO32 版 jgawk(NDP あり)
CPU Speed Check
---------------
1. for   : 5sec
2. print : 56sec
3. gsub  : 61sec
```

図 3-17 <cpuspeed.awk>の実行例（GO32 版 jgawk NDP あり）

これらの点を考え合わせると、GO32版のjgawkは、大きな連想配列が必要な場合を主に使用するのが

正解のようだ。

NDPを実装しているユーザーは、計算処理が主体の場合にも有効であるが、jgawk87版に比べて、それほど大きなアドバンテージが得られるわけではないので、MS-DOSに対するリクエストの遅さを考えるとjgawk87版の方がバランスがよい。

時間を知る

これはマニュアルにも書いてあるが、乱数の種を設定するsrand()関数を引き数を指定しないで使用すると、

現在の日時が乱数の種

として指定される。また、この関数は前回srand()関数を使用して指定した乱数の種を返す。つまり、srand()関数を引き数なしで2度呼べば、2回目のsrand()関数の返り値が現在の日時となるわけだ。

さて、srand()関数が乱数の種として使用する現在の日時の数値であるが、これは組み込み関数が宣言されているbuiltin.cの中のsrand()関数の処理を行うdo_srand()関数を見ればわかる。この関数の中をのぞいて見ると、以下のように

time()関数の戻り値

を乱数の種としているようだ。

```
srandom((int) (save_seed = time((long *) 0)));
```

そこで、以下のようにすればsrand()関数を使用して、現在の時間を文字列として返す関数を作成することができる。時間houに9を足しているのは、time()関数がGMTで時間を返すので、それをJSTに変換するためだ。

リスト3-14 現在の時間を返す関数

```
func GetTime(           sec, min, hou, tim) {
        srand()
        tim = srand()
        sec = tim % 60
        min = int(tim / 60) % 60
        hou = int(tim / 3600 + 9) % 24

        return sprintf("%02d:%02d:%02d", hou, min, sec)
}
```

意外と知られていないが、srand()で取得する現在の時刻は、環境変数TZによるタイムゾーンの指定によって変化する。まぁ、ソースコードを見れば、このあたりは容易に想像がつくが、

意外とこれにハマル

こともあるのでご注意いただきたい。ちなみに、上のスクリプトは、"set TZ=JST-9"としてキチンとタイムゾーンを日本標準時に指定した場合であり、これを行っていない場合は、コンピュータの内蔵時計をGMTとして処理するため"+9"は不要となる。

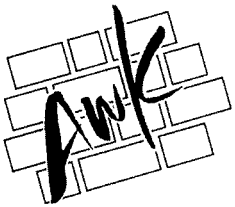
ちなみに、

gawkのver.2.13以降では

時間を知るためのsystime()と、それを文字列に変換するstrftime()という2つのシステム関数が追加されている。systime()は現在の時刻を1970

年1月1日の0時からの経過秒数で返す関数で、引数はない。

strftime(format, time)は、systime()で取得した時間を引数timeで指定すると、引数formatに指定した書式に沿って文字列に変換する関数だ(引数timeは省略可能で、その場合は、現在の時間となる)。書式に指定できる変換文字は多数にわたるので、詳しくはリファレンスを参照のこと。



検索を自由に

問題: 文章中に書かれた、「図[0–9]+」の正規表現を満たす部分を見つけだし、番号の部分を、リナンバリングするスクリプトを作りなさい。

結論: 「そんなもの、Cで書けば...」などと言っている向きは…

この節を読むように!

"RE"と/RE/の違い

awkでは、文字列定数を記述する際に、「"文字列定数"」と「/文字列定数/」の2種類の記述ができる場合がある。

この二者の違いをご存知だろうか?

実は、「/文字列定数/」の記述が許されるのは、正規表現を指定する場所なのだ。「/文字列定数/」の書式で正規表現を指定すると、内部で正規表現を表す文字列定数として認識され、静的なモノとしてテーブルの用意が行われる。これに対し、「"文字列定数"」で正規表現をした場合、変数として指定したのと同じように、動的なモノとして認識され、正規表現用のテーブルは、実行時に動的に作成される。このため、「"文字列定数"」と書くよりも、「/文字列定数/」と書いた方が、実行スピードが若干速くなる。以下、その実験のスクリプトと結果を示す。

リスト 3-15 <refix.awk>

```
BEGIN {
    i = 0
    a = "0A1BC2DEF3GHIJ4KLMNO5PQRSTUV6WXYZ7"
    f = "/dev/stderr"

    print "Regular Expression Fix String Value Speed Check"
    print "-----------------------------------------------"

    printf "1. /RE/ type     : "
    srand()
    for (i = 0; i < 30000; ++i) {
        a = "0A1BC2DEF3GHIJ4KLMNO5PQRSTUV6WXYZ7"
        gsub(/[0-9]/, "#", a)
    }
    print -srand()+srand() "sec"

    printf "2. ¥"RE¥" type     : "
    srand()
    for (i = 0; i < 30000; ++i) {
        a = "0A1BC2DEF3GHIJ4KLMNO5PQRSTUV6WXYZ7"
        gsub("[0-9]", "#", a)
    }
    print -srand()+srand() "sec"

    printf "3. Variable type : "
    b = "[0-9]"
    srand()
    for (i = 0; i < 30000; ++i) {
        a = "0A1BC2DEF3GHIJ4KLMNO5PQRSTUV6WXYZ7"
        gsub(b, "#", a)
    }
    print -srand()+srand() "sec"
}
```

<実行結果>

```
A>jgawk -f refix.awk > nul
Regular Expression Fix String Value Speed Check
-----------------------------------------------
1. /RE/ type      : 70sec
2. "RE" type      : 79sec
3. Variable type : 79sec

A>
```

図3-18 <refix.awk>の実行

コメント行をスキップする

awkのスクリプトをはじめとしてcshやshなどでも"#"以降、行末までをコメントとするテキストは多く存在する。こうしたファイルをawkで処理する場合、コメント部分に関してはスキップしたくなる。こうした場合、以下のようなスクリプトを最初のルールとして書いておけばよい。

```
/^#/ { next }
/#/ {gsub(/#.*$/, "")}
```

文章をなめる

テキストファイルに関するすべての処理をawkで行おうとしてはならない。そんなことは十分承知していても、いったんawkの環境に慣れてしまうと簡単なテキストフィルタでも、C言語で書こうという気が起こらなくなってしまう。awkには、

それだけ強い中毒性があるのだ。

たとえばawkは、ある程度決められた区切りがあり、それによってテキストをフィールド文字列の集まりとして処理できるようなケースには非常に有効なツールとなるが、こうした区切りも特になく、テキストファイルの先頭から最後までなめるように見て行かなければならないような処理の場合、スクリプトの作成が一気に難しくなる。

こうした場合に便利なのが、以下のルールだ。

リスト3-16 文章をなめるルール

```
Look["target"] {
        Look["back"] = $0
        Look["fore"] = ""

        while(match(Look["back"], Look["target"])) {
          Look["fore"]   = Look["fore"] substr(Look["back"], 1
, RSTART - 1)
          Look["find"]   = substr(Look["back"], RSTART, RLENGTH)
          Look["back"]   = substr(Look["back"], RSTART + RLENGTH)
          Look["target"] = LookCallBack(Look)
          Look["fore"]   = Look["fore"] Look["find"]
        }
        $0 = Look["fore"] Look["back"]
}
```

このルールをスクリプトの最初の方に書いておき、変数Look["target"]に検索したい文字列を指定しておく。もちろん、この変数に指定する検索文字列には正規表現を指定することも可能だ。この変数の設定は、BEGINルールで設定すればよいだろう。

このルールは、レコードのなかに検索文字列を見つけると、レコードを検索文字列前と検索結果文字列、それに検索文字列の

3つに分解

し、LookCallBackという名前のユーザー関数を呼ぶ。このコールバック関数のパラメータは1つで、Look配列が引き数となっている。Look["fore"]に検索文字列前部分，Look["find"]に検字列部分，Look["back"]に検索

文字列後部分の文字列が代入されるので，これをコールバック関数内で適当に編集すればよい．

ちなみに，コールバック関数は，検索文字列を戻り値として返さなければならない仕様になっている．これによって，

検索文字列を柔軟に変更する

ことができるようになっているわけだ．

最後に、このルールの使用例として、テキストファイルの中に散乱している図版指定に合わせて図版番号を振るスクリプトを書いてみた。

図版指定の文字列は、「図」に続けて英文字列をつなげたものとし、これの出現順に番号を振っていく。ただし以前に同じ英文字列の図版指定が使用されていた場合は、先に使用した番号を振る。

リスト3-17 <zuno.awk>

```
BEGIN {
        Look["target"] = "図[A-Za-z]+"
}

Look["target"] {
        Look["back"] = $0
        Look["fore"] = ""

        while(match(Look["back"], Look["target"])) {
          Look["fore"]   = Look["fore"] substr(Look["back"], 1
, RSTART - 1)
          Look["find"]   = substr(Look["back"], RSTART, RLENGTH)
          Look["back"]   = substr(Look["back"], RSTART + RLENGTH)
          Look["target"] = LookCallBack(Look)
          Look["fore"]   = Look["fore"] Look["find"]
        }
        $0 = Look["fore"] Look["back"]
}

{
        print $0
}
```

```
func LookCallBack(look)
{
        script = substr(look["find"], 2)
        if(script in table) {
                no = table[script]
        } else {
                no = table[script] = ++maxno
        }
        look["find"] = "図" no
        return look["target"]
}
```

<実行結果>

```
A>type zuno.txt
  図Newと図Oldを見くらべればわかるとおり、アクセルの位置が新型と旧型では若干異なることがわかる。新型(図New)では若干センター側に寄っているのだ。これはアクセルワイヤーの取り回し(図NewAccell)が、旧型のそれに(図OldAccell)くらべて滑らかになったことから実現が可能となったものだ。
```

図 3-19 編集前

では、zuno.awk を使って編集してみよう。

```
A>jgawk -f zuno.awk zuno.txt
図1と図2を見くらべればわかるとおり、アクセルの位置が新型と旧型では若干異なることがわかる。新型(図1)では若干センター側に寄っているのだ。これはアクセルワイヤーの取り回し(図3)が、旧型のそれに(図4)くらべて滑らかになったことから実現が可能となったものだ。
```

図 3-20 <zuno.awk>で編集後

結構面倒なテキストをなめるといった処理も、Look ルールを使用すると、検索した文字が見つかった時の処理を、コールバック関数として記述すればよいだけになるので、

あまり頭を使わずにスクリプトを記述できる

ことが理解して頂けただろうか。

文字列に正規表現がマッチするか?

こんな基本的な話をこんなところに書くのは、気が引けるのであるが、本書の関係者の多くから

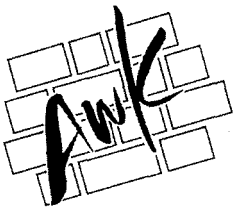
「当初、これに引っかかった」

という証言が得られたので、ここで説明しておくことにする。

この記述で注意しなければならない点は、~演算子の右と左に指定する内容が異なるという点である。平たくいえば、正規表現は必ず~演算子の右側に書かなければならないのだ。

```
式　~ /正規表現/
```

これが意外にも忘れられるケースがあるようなのだ。通常二項演算子の場合、右と左の内容は、基本的に変わらない場合がほとんどなので、つい~演算子も、どちらに正規表現を書いても大丈夫と勘違いしてしまうのだ。



レコードとフィールド

問題: awk のフィールドセパレータは動的に変更可能ですが、変数 FS を変更した場合に、いつから新しいフィールドセパレータが使用されるでしょうか？

結論: えーっ!「スクリプト実行中にフィールドセパレータを変更できることを知らなかった」⁉ そんな向きは…

この節を読むように!

レコードの区切り文字を変更する

レコードの区切り文字は、レコードセパレータと呼ばれるシステム変数 RS に設定されている。RS は初期値が改行コードになっている。もちろん、この変数もスクリプト内で再設定可能で、たとえば";"をレコードセパレータに変更することもできる。

さて、RS の再設定を行う際に、awk のスクリプトを数個書いたことのある初心者から中級者の人たちが注意しなければならない点がある。それは、レコードセパレータとして使用されるのは、RS システム変数に設定された文字列の

最初の1文字だけ

である点だ。

この点に関してはマニュアルにもキチンと説明されているのだが、フィー

ルドの区切り文字を指定するフィールドセパレータを設定するシステム変数FSが、正規表現を含む文字列として設定できるのでこれと混同してしまうことがあるようだ。ついでにレコードセパレータについて補足しておくと、システム変数RSに空文字列""を設定すると、連続した空行がレコードセパレータとなる。

入力文字列編集後、再度フィールド分解を行う

awkの仕様では$0を変更した場合、その時点で自動的に各フィールドを表す$nと、フィールド数を表すNFが再構築される。たとえばDIRコマンドの日付のフィールドを年、月、日という具合に別々のフィールドに分解したい場合は、**リスト3-18**のようにして、$0に対して文字列演算を行えば、

各フィールドの再構築が行われる。

リスト3-18 $0を変更した場合

```
{
    gsub(/-/, " ", $(NF - 1))
    $0 = $0 ""
    print
}
```

この例では、日付のフィールドだけを対象にしたいので、$0に対してgsub()関数を使用せずに（$0にgsub()関数を使うと、もしファイル名に"-"が使用されていた場合に、ファイル名が分断されてしまう可能性があるため、このような方法を取ったのだ）、最後から2番目のフィールド（つまり、日付のフィールド）にのみgsub()関数を使用して、"-"を" "に置換した。

この時点では、まだ、フィールドの再構築は行われない。フィールドの再構築が行われるのは、$0に対して演算が行われた時のみである。

ちなみに、$n に対して演算が行われた場合、

逆に$0の再構築が行われる。

この例の場合、$(NF-1) に対して gsub() 関数で文字列置換を行った時点で、$0 の文字列中の日付の"-"は" "に変更されている。

それでは、$0 に対して、数値演算を行った場合は、どのようになるかご存知だろうか？この実験を行ったのがリスト 3-19だ。意外にも$0 の値は、行頭から数値として認められる部分を数値に変換したものとなる。ここまでは何となく想像がつくが、さらに意外なことに、

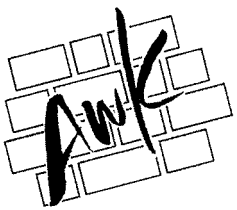
NFが0となってしまう

のである。

リスト 3-19 $0 に対して数値演算を行なった場合

```
{
    print "1) NF=" NF "$0=" $0
    $0 = $0 + 0
    print "2) NF=" NF "$0=" $0
}
```

ちなみに、FS を変更してから$0 = $0 ""などとしてフィールドの再構築を行っても、残念ながら新しい FS でフィールドでの再構築は行われず昔の FS でフィールドの再構築になってしまう。FS の変更が有効になるのは、次のレコードを読む時点からだ。



ファイルを自在に

問題: awkを使用したシェルスクリプト（バッチでも構わない）を使用している時にawkのスクリプトをsedやawkで作成し、それを標準入力から挿入したい場合がありますね。こんな時どのようにすれば良いか50字以内で明確に答えなさい。

結論: 1字も書けなかった向きは…

この節を読むように!

起動時のパラメータを知る

awkのオプションを除いて、awkの起動時のコマンドラインのパラメータは、スクリプトからARGCとARGVシステム変数を使用することによって取得することができる。パラメータと言っても実際にはawkで処理するファイル名である。

ファイル名と言えば、ワイルドカードを含んだファイル名は、UNIXの場合シェルによってワイルドカードが展開されるが、MS-DOS版ではこの仕様に合うように、awk自身がワイルドカードを含むファイル名を展開するようになっている。ただ、UNIXのシェルの展開のようにディレクトリにワイルドカードを使用したファイル名、たとえば"*/*"の展開はできないので注意してほしい。このワイルドカードの展開の様子は、以下のような簡単なスクリプトで確かめることができる。

リスト 3–20 <arg.awk>

```
BEGIN {
    for(i = 0; i < ARGC; ++i) {
        printf "ARGV[%d]=¥"%s¥"¥n", i, ARGV[i]
    }
}

A>jgawk -f arg.awk test1 test2
ARGV[0]="jgawk.exe"
ARGV[1]="test1"                ←"–f arg.awk"は awk のオプションなのでない
ARGV[2]="test2"

A>jgawk -f arg.awk *.bak       ←ワイルドカードを指定すると
ARGV[0]="jgawk.exe"
ARGV[1]="awk256_2.bak"         ←展開される
ARGV[2]="csv.bak"
ARGV[3]="time.bak"

A>jgawk -f arg.awk src¥*.*     ←ディレクトリを含んだワイルドカードの場合
ARGV[0]="jgawk.exe"
ARGV[1]="src¥alloca.c"         ←展開される。サブディレクトリファイルは展開
ARGV[2]="src¥alloca.s"          されない
ARGV[3]="src¥array.c"
    :
    :

A>jgawk -f arg.awk *.*¥*.*     ←ディレクトリ名にもワイルドカードを指定した場合
ARGV[0]="jgawk.exe"
ARGV[1]="*.*¥*.*"              ←展開されないで、そのまま
```

ARGV システム変数を変更することによって、コマンドラインで指定されたファイル名ではないファイルを読み込んだり、また新たに増やしたりまったく読まないように削除したりすることができる。具体的には、

ARGVのファイル名を""に変更すれば

そのファイルを読まなくなり、"–"とすれば、標準入力から読むようになる。

また、複数のファイルを処理する場合に現在読み込んでいるファイル名を FILENAME システム変数によって知ることができる。

リスト3-21 <filemarg.awk>

```
NowFileName != FILENAME {
    if("" != NowFileName) {
        printf "----- File '%s' End ----¥n¥n", NowFileName
    }
    NowFileName = FILENAME
    printf "----- File '%s' Start ----¥n", NowFileName
}
{
    print $0
}
END {
    if("" != NowFileName) {
        printf "----- File '%s' End ----¥n¥n", NowFileName
    }
}
```

<実行結果>

```
A>type file_a.txt
これはfile_a.txtです。
これはfile_a.txtです。

A>type file_c.txt
このメッセージは、ファイルに書かれたものだ。
ちなみに、本ファイルはfile_c.txtです。

A>jgawk -f filemarg.awk file*.txt
----- File 'file_a.txt' Start ----
これはfile_a.txtです。
これはfile_a.txtです。
----- File 'file_a.txt' End ----

----- File 'file_b.txt' Start ----
こいつがfile_b.txtだ。
----- File 'file_b.txt' End ----
```

```
----- File 'file_c.txt' Start ----
このメッセージは、ファイルに書かれたものだ。
ちなみに、本ファイルはfile_c.txtです。
----- File 'file_c.txt' End ----

A>
```

図 3-21 <filemarg.awk>の実行例

入力ファイルを変更する

プログラムの中には、ファイルを指定する際に、ファイル名の拡張子を省略できるものがある。この例としては compass や cc、as など数え上げればキリがない。

awk でデータを処理する場合も、このような機能を awk の中で実現することができる。ARGV 変数を BEGIN ルールの中で変更してしまえばよいのだ。以下のスクリプトは、データファイルとして指定されたファイル名に拡張子が指定されていない場合、自動的に".awk"の拡張子を付けてそのファイルを出力するものだ。

リスト 3-22 <prnawk.awk>

```
BEGIN {
    for(loop = 1; loop < ARGC; ++loop) {
        if(! match(ARGV[loop], /¥.[^.¥¥]*$/)) {
            ARGV[loop] = ARGV[loop] ".awk"
        }
    }
}

{
    print
}
```

<実行結果>

```
A>dir

 ドライブ B: のボリュームラベルは SD01C
 ボリュームシリアル番号は 190D-12EE
 ディレクトリは B:¥HOME¥TAKUYA¥AWK¥TMP

.            <DIR>      93-01-31   14:31
..           <DIR>      93-01-31   14:31
prnawk   awk        155 93-02-28   22:03
test                 23 93-02-28   22:01
test     awk         40 93-02-28   22:01
        5 個              218 バイトのファイルがあります.
                      2969600 バイトが使用可能です.
```

図3-22 サンプルデータ

<実行結果>

```
A>jgawk -f prnawk.awk test
BEGIN {
    print "This is TEST.AWK"
}

A>jgawk -f prnawk.awk test.
This is "TEST" file.

A>
```

図3-23 <prnawk.awk>の実行

この他にも、引数としてファイル名が指定されていない場合に既定のファイルを読むようにするようなことも、このARGVとARGCを変更することで可能だ。

リスト 3-23 <deffile.awk>

```
BEGIN {
    if(ARGC < 2) {
        ARGC = 2
        ARGV[1] = "default.txt"
    }
}
{
    print
}
```

<実行結果>

```
A>type default.txt
This is default file.

A>jgawk -f deffile.awk
This is default file.

A>
```

図 3-24 <deffile.awk>の実行例

ARGV の変更は、BEGIN ルールのなかでなくても変更できる。しかし、すでに読み込みが始まっているファイルについてはいくら ARGV を変更しても読み込むファイルは変更できないので注意すること。たとえば、コマンドラインに 2 つのファイルが指定されていて、1 つ目のファイルを読み込んでいる時に、2 つ目のファイル名を変更することは可能だ。また前の例を応用して、ARGC と ARGV を変更することでスクリプトのなかで処理するファイルを追加するようなことも可能だ。

スクリプトを標準入力から指定したい

あまり一般的な利用形態ではないが、awkのスクリプトを標準入力から指定することもできる。"-f -"として、スクリプト名を"-"にすると、標準入力となる。

```
awk -f - <データファイル名>
```

たとえば、awkスクリプトをm4やcppなどのマクロプロセッサにとおしてからawkで実行するような場合に利用できる。とは言っても、双方のマクロプロセッサをともに"$"の扱いにおいてそのままawkのスクリプトに応用するのは難しいし、m4に至っては、パソコンで動く処理系を入手するのは困難だろう。

そこでサンプルとして、以下のような簡単な置換を行なうawkスクリプトを作ってみた。内容については、スクリプトを読んで頂ければ、一目瞭然だろう。

リスト3-24 <awkpp.awk>

```
1 < NF && "#define" == $1 {
    tbl[$2] = $3
    next
}

{
    for(define in tbl) {
        gsub(define, tbl[define])
    }
    print $0
}
```

この置換スクリプトを使用してawkのスクリプトを置換し、その結果をawkのスクリプトとして実行するような実験を行ってみた。

リスト 3-25 <dirawk.awk>

```
#define ファイル名  $1
#define 拡張子      $2
#define サイズ      $3
#define が          ==
#define に加算      +=
#define の時        {
#define とする      }
#define 出力        print
#define 最後        END
#define 合計        total

拡張子 が "awk" の時
    合計 に加算 サイズ
    出力 ファイル名 "." 拡張子
とする

最後 の時
    出力 "( " 合計 "byte )"
とする
```

<実行結果>

```
A>type dir.lst

 ドライブ B: のボリュームラベルは SD01C
 ボリュームシリアル番号は 190D-12EE
 ディレクトリは B:¥HOME¥TAKUYA¥AWK¥FIX

.            <DIR>      93-01-06    1:07
..           <DIR>      93-01-06    1:07
ascmail  awk        645 92-05-22    0:00
ascmail  bat         52 92-05-17    2:27
awk256_2 80       17918 93-01-06    0:58
awk財務  lzh       7113 92-12-15    1:14
esc      awk      10193 92-10-26   22:58
memo1    80        1735 92-12-29   16:44
memo2    80        1812 92-12-30    4:03
memo3    80        2765 93-01-07    0:15
```

```
mixmail  awk       722 92-05-17    2:16
mixmail  bat        52 92-05-17    2:01
numnis   awk       993 92-12-27   22:39
sample   doc       552 92-12-15   23:02
uniqnis  awk       311 92-12-27   17:42
uniqnis2 awk      2501 92-12-29   17:20
virus    awk       726 93-01-19   23:56
英和     awk      2398 92-12-23   17:17
英和     lzh      1824 92-12-16    0:19
英和2    lzh      1499 92-12-17    0:50
勘定科目          1399 92-06-03   22:51
国語     awk      2275 92-12-17    0:44
財務会計 awk     12459 92-12-15    1:13
振替伝票 91      14294 92-12-15    1:12
       24 個           84238 バイトのファイルがあります.
                     3416064 バイトが使用可能です.
```

図 3-25 <dir.lst>

＜実行結果＞

```
A>jgawk -f awkpp.awk dirawk.awp | jgawk -f - dir.lst
ascmail.awk
esc.awk
mixmail.awk
numnis.awk
uniqnis.awk
uniqnis2.awk
virus.awk
英和.awk
国語.awk
財務会計.awk
( 33223byte )

A>
```

図 3-26 <dirawk.awk>の実行例

とやってみたものの、スクリプトの生成にも時間がかかるし標準入力からデータを入れられないし、ということで、

うーん、実際にはあまりいいことはない

(というかこの例は、あまり適切な例ではなかったということか)。しかし、後々必要となる局面に遭遇しないとも限らない。覚えておくにこしたことはないだろう。

スクリプトの保管場所を作る

UNIXで、#!構文が使用できる環境では、awkのスクリプトも、パーミッションを0755などにし、

"#! /usr/local/bin/gawk -f"などとする

ことで、シェルスクリプトと同じように1つの実行可能ファイルとして扱うことができるようになるのでPATHの設定されているディレクトリに置いておけば具合がよい。

残念ながらこの構文が使用できない環境では、作成したawkのスクリプトは

どこに置いておけばよいのか

迷うところだ。gawkではAWKPATHによってawkのスクリプトを探すディレクトリを指定できるので便利だ。たとえばawkのスクリプトを/usr/lib/awkなどの適当なディレクトリに集めておいて、環境変数AWKPATHにそのディレクトリを設定しておけば、awkのスクリプトを指定する際に

いちいちフルパスで指定しなくても

ファイル名を指定するだけでgawkが探し出してくれる。

環境変数を取得する

awkのスクリプトから、シェルの保持する環境変数を参照しようとする場合には、ENVIRONシステム変数を利用する。このシステム変数は、環境変数名が添え字の連想配列になっているので、たとえば環境変数HOMEの内容をhomeDir変数に代入するには、

```
"homeDir = ENVIRON["HOME"]"
```

とすればよい。

注意しなければならない点は、このENVIRONシステム変数の内容をスクリプトの中から設定し直しても、その値がシェルの環境変数に設定されることは一切ないという点だ。これは、awkを起動したシェルはもちろんのこと、system()組み込み関数やパイプによってスクリプト内から起動するプログラムの環境変数としても、まったく使用されない。

こんなことも…

問題: awkは、しばしば他のコマンドの出力結果を処理することがありますね。こういった場合、通常ではパイプ機能を利用して、2つのコマンドを連結しますが、シェルにパイプ機能が付いていない場合、どのようにすればよいでしょうか？

結論: bashやtcshをコンパイルすると答えた向きは…

この節を読むように!

対話形式のスクリプトを作る

必要に応じて、ユーザーに条件の入力を求めるようなスタイルを

対話形式という。

awkでもgetlineコマンドを使えば、簡単に対話形式のプログラムを作成することができる。たとえば、ある文字列を指定した文字列に置換するスクリプトを対話形式で行なうようにする場合は、以下のようにしてgetlineコマンドを使用する。

リスト3-26 <reps.awk>

```
BEGIN {
    printf "対話型文字列置換¥n"                    > "/dev/stderr"
    printf "検索文字列を入力してください: "        > "/dev/stderr"
    getline old                                    < "/dev/stdin"
```

```
    if("" == old) exit
    printf "置換文字列を入力してください: "    > "/dev/stderr"
    getline new                                < "/dev/stdin"
    if("" == new) exit
}
{
    gsub(old, new)
    print
}
```

<実行結果>

```
A>type reps.old
  明日は、楽しい衆議院選挙です.
なのに私は、南青山で徹夜です. シ
クシクシク. あーぁ、阿部チャンは
今ごろ何をしているのかなぁ...

A>jgawk -f reps.awk reps.old > reps.new
対話型文字列置換
検索文字列を入力してください: .
置換文字列を入力してください: 。

A>type reps.new
  明日は、楽しい衆議院選挙です。
なのに私は、南青山で徹夜です。シ
クシクシク。あーぁ、阿部チャンは
今ごろ何をしているのかなぁ...
```

図3-27 <reps.awk>の実行

getlineコマンドに「<"/dev/stdin"」を付けて、強制的に標準入力から取得しているところがミソである。こうしないと、データファイルから1レコード分データを取得し、それがgetlineコマンドの後ろに書かれている変数に代入されてしまう。

ちなみに、「検索文字列を入力してください: 」などの文字列出力を「>"/dev/stderr"」に向けているのは、図3-27のように、本スクリプトを実行する際に置換結果をMS-DOSのリダイレクションを利用してファイルに取得する時に、余分なメッセージがファイルに出力されないように

するとともに、必ず画面に表示されるようにするためである。

外部コマンドの実行結果を得る

たとえば、

DIRコマンドの出力結果をawkで処理する

ような場合、あなたならどうするだろう？ 普通に考えれば、以下のようにDIR コマンドと awk コマンドをパイプでつないで DIR コマンドの出力を awk に喰わせるようにするのではないだろうか？

```
A>dir | jgawk -f script.awk
```

この方法は、入力が DIR コマンドでなくてもよいといった柔軟性があるし、データとして入力されるため、フィールド分解が自動的に行われるのでスクリプトが記述しやすいといった利点がある。

しかし、この方法は実行スピードが遅い。UNIX ではパイプで連結された複数のコマンドは別々のプロセスとして起動され、それぞれの標準入出力を IPC（インタープロセス・コミュニケーション）によって接続するのでそれほどでもないのだが、MS-DOS は

悲しいかなシングルタスク

の OS なのでこうしたことができない。MS-DOS でパイプを使用すると、通常はディスク上に一時ファイルを作成し、1つのコマンドが終わると次のコマンドの標準入力にその一時ファイルの内容を流し込むといったことが行われてしまう。なんとも「ダサダサ」だ。まったく！

そこで、MS-DOS でスピードを追求するために、getline コマンドを利用してコマンドの実行結果をダイレクトに読み取ってみる。これならひょっとすると一時ファイルを作らずに実行できるかも知れない。

たとえばDIRコマンドの出力結果から、ディレクトリファイル以外のファイルを抜き出すといった処理を行うような場合、以下のようなスクリプトとなる。

リスト3-27 <DIR>からファイルを抜き出す

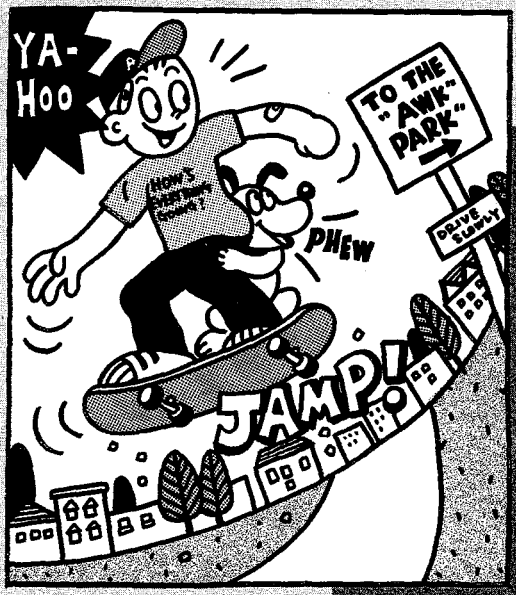
```
BEGIN {
    while("dir /s" | getline) {
        if("ディレクトリは" == $1) {
            print $2
        } else if("バイトのファイルがあります." == $4) {
            print ""
        } else if(0 != NF && $1 !~ /¥./) {
            print
        }
    }
}
```

さて実行スピードであるが、これが予想に反して結構時間がかかる。どうやら一度コマンドを実行し終わってからその結果をgetline関数に渡しているようだ。つまり、パイプを使用するのとなんら変わるところがない。このような記述はコマンドの起動を固定した形でスクリプトに内蔵してしまいたい場合などを除けば、

あまり意味はないようである。

awk 実践プログラミング

—100行野郎の実際—



新しいプログラミング言語を覚える時に最も参考になるのは、他人の書いたプログラムであることは周知の事実である。そこで本書もサンプルのawkのスクリプトを用意した。これらのサンプルは、awkのスクリプトを書く際に参考になることはもちろん、それ単体でも結構楽しめるユニークなものが揃っている。ぜひリストを打ち込んで、スクリプトを実行してみてほしい。

超かんたんテキスト DBMS!!

awk 住所録

機　能

テキストファイルで用意した住所録データを読み込み、検索するスクリプトだ。住所録データのフォーマットの変更やデータの出力形式をスクリプトに手を入れることなしに実現できる。つまり扱うデータが住所録でなくても、いっこうにかまわないのだ。スクリプトのタイトルこそ awk 住所録となっているが、awk を使用した簡単な DBMS といった感じだ。

操　作

まず "addr.db" という名前で住所録ファイルを用意する (図 4–1)。フィールドセパレータは ":"、レコードセパレータは改行で、"ふりがな"、"名前"、"郵便番号"、"住所"、"電話番号"、"勤務先"、"備考" の順でフィールドを用意する。エディタなどを使用してテキストファイルとして用意すればよい。ちなみに行頭が "#" で始まる行は注釈として無視される。

```
ふじ　えいじ:藤　英二::葛飾区柴又 XX-X-XX:03-3:(宗)明日来教団:
しむら　たく:志村　拓:176:練馬区豊玉北 X-X:3X-XX:世界征服(有)
なかべ　ただし:中部　正:184:町田市柄割 X-X:3XX-XXXX:(株)印食:
おかべ　かず:岡部　和成:610:大阪市雨野宝前寺 X-X:XX-XXX:トラ吉
```

図 4–1　入力データ例

この住所録ファイルを検索するには本スクリプトのデータとして指定すればよい。スクリプトを実行すると画面が図4–2のようになる。最後に表示されている“AWKaddr>”が本スクリプトのプロンプトである。この状態で、その上の行に表示されている各メニュー項目の左に書かれている英文字のコマンドを入力することによって、対話形式で作業を進めていく。以下にそれぞれのコマンドについて説明する。

```
A>jgawk -f addr.awk addr.db
awk住所録
Copyright (C) 1993 by Taku-SHIMURA

住所録ファイル: addr.db
データ読込中... 58件

S)検索 A)追加 E)編集 D)削除 P)表示 W)更新 Q)終了

AWKaddr>
```

図4–2　<addr.awk>の実行例

S)検索

“S”コマンドは、住所録から目的のデータを検索するためのコマンドだ。本コマンドを実行すると、画面に1レコード分のデータを入力するためのフォームが表示される。ここで、入力した各フィールドに一致したレコードが検索される。各フィールドの指定には正規表現を使用することが可能になっている(図4–3)。ちなみにフィールドの入力のところで、“^”だけを入力してやると前のフィールドにカーソルを戻すことができる。

```
AWKaddr> s
レコードの検索
ふりがな:
名　　前:
郵便番号:
住　　所: ^東京
```

```
電話番号:
勤 務 先:
備    考:
確認(Yes,No,Cancel): y
6件
AWKaddr>
```

図 4–3　S) コマンドの使用例

このコマンドは検索結果として何件のデータが検索できたかのみを表示する。検索結果の内容を表示したい場合は、後で説明する “P) 表示” コマンドを使用する。

A)追加

“A” コマンドは、住所録データに新たにデータを追加するコマンドだ。本コマンドを実行すると “S” コマンドと同じように、1 レコード分のデータを入力するためのフォームが表示されるので追加するデータを入力する。ここでもフィールドの入力のところで、“^” だけを入力してやると前のフィールドにカーソルを戻すことができる (図 4–4)。

```
AWKaddr> a
レコードの追加
ふりがな: わしきた　けん
名    前: 鷲北　賢
郵便番号:
住    所:
電話番号:
勤 務 先:
備    考:
確認(Yes,No,Cancel):
```

図 4–4　A) コマンドの使用例

本コマンドで追加したデータは、後で説明する “W) 更新” コマンドを実行することによって住所録ファイルに更新される。これを忘れるとせっかく入力したデータもメモリの狭間に消えてしまう。

E)編集

"E" コマンドは "S" コマンドで検索したデータを編集修正するコマンドだ。本コマンドを実行すると、検索されているデータが次々と表示されるので修正したいフィールドで文字列を入力する。前と同じでよいフィールドはリターンキーのみを入力すればよい。

```
AWKaddr> e
ふりがな: しむら　たく
名　　前: 志村　拓
郵便番号: 176
住　　所: 東京都練馬区豊玉北 X-X
電話番号: 03-39XX-XXXX
勤 務 先: 世界征服(有)
備　　考:
確認(Yes,No,Cancel): y
```

図 4-5　E) コマンドの使用例

D)削除

"D" コマンドは、"S" コマンドで検索したデータをデータベースから削除するコマンドだ。本コマンドを実行すると、本当にデータを削除してしまってよいか聞いてくるのでよければ "Y" を入力する。

```
AWKaddr> d
消去確認(Yes,No): n
キャンセル
AWKaddr> d
消去確認(Yes,No): y
8件削除
AWKaddr>
```

図 4-6　D) コマンドの使用例

本コマンドも前出の“A”コマンドと同じように、後で説明する“W)更新”コマンドを実行することによって住所録ファイルに更新される。

P)表示

“P”コマンドは、“S”コマンドで検索したデータを画面または指定したファイルに出力するコマンドだ。本コマンドを実行すると出力先を聞いてくるので、ファイルに出力したい場合はここでファイル名を指定する。ここで何も入力しないと画面に表示される(図4–7)。

```
AWKaddr> p
出力先:

氏名: 藤　英二（ふじ　えいじ）
住所: 〒 東京都葛飾区柴又 XX-X-XX
TEL.: 03-3
備考:（勤務先（宗）明日来教団）

氏名: 志村　拓（しむら　たく）
                :
                :
TEL.: 06-XXX-XXX
備考:（勤務先（株）トラ吉）

--More-- (50%)
```

図4–7　P)コマンドの使用例

データを画面に表示した場合、データが1画面に収まらなくなった時点で“--More-- (??%)”と表示され、出力が止まる。次のデータを見る場合にはリターンキーを押す。ここで“Q”を入力してからリターンキーを入力すると、表示の途中で本コマンドを終了することができる。

W)更新

“W”コマンドは、“A”コマンドや“D”コマンドで住所録のデータを追加・削除した結果を住所録ファイルに更新するコマンドだ。本コマンドを実行すると出力先を聞いてくるのでここで現在使用している住所録ファイ

ルとは違うファイルに現在のデータを出力したい場合はそのファイル名を指定する。ここで何も指定しないと現在使用している住所録ファイルを更新する。

```
AWKaddr> w
出力先:

AWKaddr>
```

図 4–8　W) コマンドの使用例

本コマンドを実行すると、system 関数を使用して住所録ファイルの BAK ファイルを作成する。この時、場合によっては system 関数で発行した MS–DOS のコマンドがエラーとなりそのメッセージが表示されてしまう。これに関してはとりあえず無視していただいて結構だ。

Q)終了

"Q" コマンドは、本スクリプトを終了するためのコマンドだ。"A" コマンドや "D" コマンドでデータを追加・削除したにも関わらず、"W" コマンドで住所録ファイルを更新していない場合に本コマンドを実行すると、本当にデータを更新しないで終了してよいか確認するメッセージが表示される (図 4–9)。ここで、"Y" を入力すると、住所録ファイルを更新することなく強制的に本スクリプトを終了する。ちなみに住所録ファイルの更新が必要でない場合は本コマンドを実行した時点で、何も表示しないでいきなり本スクリプトを終了する。

```
AWKaddr> q
データが更新されていません。
終了確認(Yes,No): y

A>
```

図 4–9　Q) コマンドの使用例

解　説

住所録ファイルの既定値

本スクリプトでデータファイルを指定しなかった場合、まず環境変数 ADDRDIR に設定されたディレクトリにある “addr.db” ファイルを使用する。環境変数 ADDRDIR が設定されていなかった場合、次に環境変数 HOME に設定されたディレクトリにある “addr.db” ファイルを使用する。環境変数 HOME も設定されていなかった場合、最後にルートディレクトリにある “addr.db” ファイルを使用する。これらのいずれにも “addr.db” ファイルが存在しなかった場合は、残念ながら awk のエラーとなり、スクリプトは異常終了する。

レコードの変更

最初のところで書いたとおり、本スクリプトは、スクリプトを変更することなしにレコードのフィールド構成を変更することができる。これを行うのがデータファイルの中に書かれた、“#RecDetail:” プロパティだ。この後にそのデータファイルに格納されているレコードのフィールド構成を “:” で区切って記述しておけば、本スクリプトがそれを読み、その指定に従って対話形式の各メッセージを変更する。

別の用途のデータベースとして使う

例えばスケジュール管理のデータベースにするのであれば、以下のような感じで、“#RecDetail” 命令を指定して、“日時”、“時間”、“場所”、“内容” の順でフィールドが並んでいることを本スクリプトに教えてやればよい。もちろん、その後に記述するデータもこの順に書いて用意しておく。あとは住所録の時と同じようにデータファイルを本スクリプトに喰わせてやれば、フィールドの指定も “#RecDetail:” プロパティで指定したとおりになってくれる (**リスト 4–1**)。

リスト4-1　スケジュール管理

```
#
#awk住所録 Data file
#
#RecDetail:日付:時間:場所:内容
#PrnDetail:$1$ $2$ $4$ ($3$)
#PrnDetail:
#
05/06/04:1930:大仁堂ビル4F:月刊ASCII PlayRite連載打ち合わせ
05/06/10:1930:吉岡ビル:月刊UPLINK DOS連載打ち合わせ
05/06/18:1930:大仁堂ビル6F:awk256倍打ち合わせ
```

では使用例を見て欲しい。

```
A>jgawk -f addr.awk sced.db
awk住所録
Copyright (C) 1993 by Taku-SHIMURA

住所録ファイル: sced.db
データ読込中... 1件2件3件

S)検索 A)追加 E)修正 D)削除 P)表示 W)更新 Q)終了

AWKaddr> s
レコードの検索
日付:
時間:
場所: 大仁堂
内容:
確認(Yes,No,Cancel): y
2件一致
AWKaddr> p
出力先:

05/06/04 1930 月刊ASCII PlayRite連載打ち合わせ (大仁堂ビル4F)

05/06/18 1930 awk256倍打ち合わせ (大仁堂ビル6F)

AWKaddr> q

A>
```

図4-10 スケジューラとしての実行例

表示の変更

ちなみにレコードの内容を変更した場合、“P) 表示” コマンドの表示も変更しないと表示内容と実際のデータの中身がちぐはぐになってしまう。この “P” コマンドの表示フォームを変更するのが、“#PrnDetail:” プロパティだ。

データファイルのなかに “#PrnDetail:” に続けて、“P” コマンド実行時に表示したい文字列を記述する。この時各フィールドの値を “$フィールド番号$” として指定することができる（awk のスクリプトが “$フィールド番号” で指定するのに対して、“$フィールド番号$” とフィールド番号の後ろにも “$” が必要な点に注意すること）。また、“#PrnDetail:” は複数書くことができる。この場合、ファイルの先頭から順番に表示される。“P” コマンドの出力を複数行にしたい場合はこの方法を使用する。

先のリスト 4–1に、実際の “PrnDetail:” プロパティの指定例とその実行結果が表示されているのでこれを見れば理解できるはずだ。

問題点

本スクリプトは全データを連想配列に保存している。このため検索のスピードは速いのだが、その反面メモリを馬鹿喰いする。データが多くなるとけっこう厳しいかもしれない。その場合は GO32 版を使用して... またデータ変更後は、“w) 更新” しないと修正内容が失われてしまうのだが、スクリプト自体が Ctrl–C によっていとも簡単に終了してしまう。

スクリプト

awk住所録

```
#
# AWK住所録 version 1.1
# Copyright (C) 1993 by Taku-SHIMURA
#
# 簡単な住所録ＤＢＭＳです.
# [on Japanized Gnu Awk (jgawk) 2.15.2 + 1.0(SJIS)]


BEGIN {
    FS = OFS = ":"

    ScrnHeight = 25      # 画面の行数

    EscSeqFmt["move"]  = "¥033[%d;%dH"
    EscSeqFmt["attr"]  = "¥033[%dm"
    EscSeqFmt["clr"]   = "¥033[2J"
    EscSeqFmt["save"]  = "¥033[s"
    EscSeqFmt["rest"]  = "¥033[u"
    EscSeqFmt["up"]    = "¥033[%sA"
    EscSeqFmt["down"]  = "¥033[%sB"
    EscSeqFmt["right"] = "¥033[%sC"
    EscSeqFmt["left"]  = "¥033[%sD"
    EscSeqFmt["clreol"] = "¥033[K"

    Menu = "{S}}検索 {A)}追加 {E)}修正 {D)}削除 {P)}表示 {W)}更新 {Q)}終了"
    gsub(/{/, "¥033[33m", Menu)
    gsub(/}/, "¥033[36m", Menu)
    YorN = "({Y}es,{N}o)"
    gsub(/{/, "¥033[33m", YorN)
    gsub(/}/, "¥033[36m", YorN)
    YorNorC = "({Y}es,{N}o,{C}ancel)"
    gsub(/{/, "¥033[33m", YorNorC)
    gsub(/}/, "¥033[36m", YorNorC)

    if(1 == ARGC) {
        if("" == (file = ENVIRON["ADDRDIR"])) {
            file = ENVIRON["HOME"]
        }
        sub(/¥¥$/, "")
        file = file "¥¥addr.db"
        ARGV[1] = file
```

```
    }
    ARGC = 2

    printf "AWK住所録¥n"
    printf "Copyright (C) 1993 by Taku-SHIMURA¥n"
    printf "¥n"
    printf "住所録ファイル: %s¥n", ARGV[1]
    printf "データ読込中... "
    EscSeq("save")          }

$0 !~ /^#/{
    Data[++RecNo] = $0
    Slct[RecNo]   = 1
    ++SlctNo
    EscSeq("rest")
    printf "%d件", RecNo        }

"#RecDetail" == $1 && 1 < NF {
    RecDtl = $2
    for(fldno = 3; fldno <= NF; ++fldno) {
        RecDtl = RecDtl FS $fldno
    }       }

"#PrnDetail" == $1 && 1 < NF {
    ++PrnDtlNo
    PrnDtl[PrnDtlNo] = $2
    for(fldno = 3; fldno <= NF; ++fldno) {
        PrnDtl[PrnDtlNo] = PrnDtl[PrnDtlNo] FS $fldno
    }       }

END {
    if("" == RecDtl) {
        RecDtl = sprintf("ふりがな%s名    前%s郵便番号%s住    所
%s電話番号%s勤 務 先%s備    考", FS, FS, FS, FS, FS, FS)
    }
    FldDtlNo = split(RecDtl, FldDtl)
    if("" == PrnDtlNo) {
        PrnDtlNo            = 0
        PrnDtl[++PrnDtlNo] = "氏名: $2$ ($1$)"
        PrnDtl[++PrnDtlNo] = "住所: 〒$3$ $4$"
        PrnDtl[++PrnDtlNo] = "TEL.: $5$"
        PrnDtl[++PrnDtlNo] = "備考: (勤務先 $6$) $7$"
        PrnDtl[++PrnDtlNo] = ""
    }
    if("" == ScrnDtl) {
        ScrnDtl = 24
    }
```

```
    printf "¥n¥n"
    print Menu
    printf "¥n"
    while(1) {
        printf "AWKaddr> "
        getline cmd < "/dev/stdin"
        cmd = toupper(cmd)
        if("" == cmd) {
        } else if("S" == cmd) {
            RecSearch()
        } else if("A" == cmd) {
            RecAdd()
        } else if("E" == cmd) {
            RecEdit()
        } else if("D" == cmd) {
            RecDelete()
        } else if("P" == cmd) {
            RecPrint()
        } else if("W" == cmd) {
            RecWrite()
        } else if("Q" == cmd) {
            if(ModData) {
                printf "データが更新されていません．¥n"
                if("Y" == Sure("終了", YorN)) {
                    break;
                }
            } else {
                break;
            }
        } else {
            EscSeq("clr")
            print Menu
            print ""
        }
    }
    EscSeq("attr", 0)
}

# 対話形式で入力された1レコード分のデータを、住所録データベースのデータと
# 追加登録する。
func RecAdd(    rec, fld, nulfld)
{
    nulfld = "^" FS "*$"
    while(1) {
        print "レコードの追加"
        rec = FldGet("")
        if(rec ~ nulfld) {
```

```
            break
        }
        Data[++RecNo] = rec
        ModData = 1
        Slct[RecNo] = 1
        ++SlctNo
    }   }

# レコードの検索を行い、その結果をSlct配列に格納する。当初は連想配
# 列を使用して検索を行う予定であったが、その方法だと検索文字列とし
# て正規表現が使用できないので、1レコードづつループを回している。
# まぁ、本物のＤＢＭＳでもインデックスを使用しての検索は、完全一致
# でしか検索できないものが多いんだからしょうがない。
func RecSearch(     rec, fn, fld)
{
    print "レコードの検索"
    srec = FldGet("")
    if("" != srec) {
        split(srec, sfld)
        SlctNo = 0
        for(rec in Data) {
            split(Data[rec], fld)
            for(fn = 1; fn <= 7; ++fn) {
                if("" != sfld[fn] && fld[fn] !~ sfld[fn]) {
                    break
                }
            }
            if(7 < fn) {
                Slct[rec] = 1
                ++SlctNo
            } else {
                delete Slct[rec]
            }
        }
        printf "%s件一致¥n", SlctNo
    }
}

# Slct変数に格納されているレコードのデータを画面に出力する。
func RecPrint(      rec, out, more, count, ans, moremax)
{
    printf "出力先: "
    getline out < "/dev/stdin"
    print ""
    if(out ~ "^[ ]*$") {
        out = "/dev/stdout"
        more = 1
```

```
    } else {
        more = 0
    }

    count   = 0
    moremax = int((ScrnHeight - 2) / PrnDtlNo)
    for(rec = 1; rec <= RecNo; ++rec) {
        if(rec in Slct && 1 == Slct[rec]) {
            FldPrint(Data[rec], out)
            ++count
            if(more && count % moremax == 0) {
                EscSeq("attr", 33)
                printf "--More-- (%d%%)", count / SlctNo * 100
                EscSeq("attr", 36)
                getline ans < "/dev/stdin"
                EscSeq("up")
                EscSeq("clreol")
                ans = toupper(substr(ans, 1, 1))
                if("Q" == ans) {
                    break
                }   }   }   }
    if("/dev/stdout" != out) {
        close(out)
    }   }

# Slct変数に格納されているレコードのデータを修正する。
func RecEdit(       recno, rec, count)
{
    count = 0
    for(recno = 1; recno <= RecNo; ++recno) {
        if(recno in Slct && 1 == Slct[recno]) {
            if((rec = FldGet(Data[recno])) !~ /^:*$/) {
                Data[recno] = rec
                ++count
                ModData = 1
            }
        }
    }
    printf "%d件修正¥n", count
}

# Slct変数に格納されているレコードのデータを住所録データベースから
# 削除する。
func RecDelete(     rec, count)
{
    if("Y" != Sure("消去", YorN)) {
        print "キャンセル"
```

```
        return
    }
    count = 0
    for(rec = 1; rec <= RecNo; ++rec) {
        if(rec in Slct && 1 == Slct[rec]) {
            delete Data[rec]
            delete Slct[rec]
            ++count
            ModData = 1
        }
    }
    SlctNo = 0
    printf "%d件削除¥n", count
}

# 現在保持している住所録データを，住所録データファイルに格納する．
func RecWrite(      file, backfile)
{
    printf "出力先: "
    getline file < "/dev/stdin"
    print ""
    if(file ~ "^[ ]*$") {
        file = FILENAME
    }

    bakfile = file
    sub(/¥.[^.]*$/, "", bakfile)
    bakfile = bakfile ".bak"
    system(sprintf("del %s", bakfile))
    system(sprintf("ren %s %s", file, bakfile))

    print "#"                          >  file
    print "#AWK住所録 Data file"                  >> file
    print "#"                          >> file
    print "#RecDetail", RecDtl                >> file
    for(rec = 1; rec <= PrnDtlNo; ++rec) {
        print "#PrnDetail", PrnDtl[rec]          >> file
    }
    print "#"                          >> file
    for(rec = 1; rec <= RecNo; ++rec) {
        if(rec in Data) {
            print Data[rec]                >> file
        }
    }
    if("/dev/stdout" != file) {
        close(file)
    }
```

```
    ModData = 0
}

# 引数recで指定したレコードの内容を規定値として，1レコード分のデータを
# 対話形式で取得する関数．取得したデータはレコードの形で，文字列として
# 戻り値となる．
func FldGet(rec,    fld, ans, ret, fldno)
{
    split(rec, fld)
    for(fldno = 1; fldno <= FldDtlNo; ++fldno) {
        printf "%s: %s¥n", FldDtl[fldno], fld[fldno]
    }
    print ""
    EscSeq("up", FldDtlNo + 1)
    while(1) {
        fldno = 1
        while(1 <= fldno && fldno <= FldDtlNo) {
            EscSeq("right", length(FldDtl[fldno]) + 2)
            EscSeq("save")
            EscSeq("attr", 33)
            printf "%s", fld[fldno]
            EscSeq("rest")
            getline ans < "/dev/stdin"
            EscSeq("rest")
            EscSeq("clreol")
            EscSeq("attr", 36)
            if("^" == ans || " ^ " == ans) {
                print fld[fldno]
                EscSeq("up", 2)
                --fldno
            } else {
                if("" != ans){
                    fld[fldno] = ans
                }
                print fld[fldno]
                ++fldno
            }
        }
        if(fldno < 1) {
            EscSeq("down", FldDtlNo + 1)
            print "キャンセル"
            print ""
            return ""
        } else {
            ret = fld[1]
            for(fldno = 2; fldno <= FldDtlNo; ++fldno) {
                ret = ret FS fld[fldno]
```

```
            }
        }

        ans = Sure("", YorNorC)
        if("C" == ans) {
            ret = ""
            break
        } if("Y" == ans) {
            break
        }
        EscSeq("up", FldDtlNo + 1)
    }
    return ret
}

# 引数recで指定した，レコードを画面に出力する関数
func FldPrint(rec, file,    fld, dltno, prn, prnno)
{
    split(rec, fld)
    for(dltno = 1; dltno <= PrnDtlNo; ++dltno) {
        prn = split(PrnDtl[dltno], prn, "$")
        for(prnno = 1; prnno <= prn; ++prnno) {
            if(prn[prnno] ~ /[0-9]+/) {
                printf "%s", fld[prn[prnno]]    >> file
            } else {
                printf "%s", prn[prnno]     >> file
            }
        }
        printf "¥n"                 >> file
    }
}

# 確認を行う関数
func Sure(msg, item,    ans)
{
    printf "%s確認%s: ", msg, item
    getline ans < "/dev/stdin"
    return toupper(substr(ans, 1, 1))
}

# エスケープシーケンスを発行する関数
func EscSeq(cmd, arg1, arg2)
{
    printf EscSeqFmt[cmd], arg1, arg2
}
```

正規表現はどこだ!

ここだよーんゲーム

機　能

「何度本を読んでも正規表現はよう解らん。」といった向き、「本を読んで覚えるくらいなら、そんなもんいらなーい」といった向きのために、正規表現をゲームにしてみた。ゲームを楽しみながら正規表現をマスターできるという、ほんとうにそうなら画期的なスクリプトだ（といってもなんか説教臭くていまいち楽しめないかもしれない）。

操　作

本スクリプトを実行する前にいくつか用意しなければならないものがあるので、まずそれらについて説明する。

1つは、"BATKEY.COM" というプログラムだ。MS-DOS Ver.5.x から用意されたプログラムで、こいつを PATH の利いたディレクトリに用意しておく。一応、Ver.5.x 以前の MS-DOS を使用している向きは、だまされたと思ってエディタなどで、

エ８５００へ！エLへ！

という文字列（ちなみにすべて半角1バイト文字）を BATKEY.COM というファイルで作成してみよう。もし間違いがなければ、BATKEY.COM

としてまっとうに動くはずだ。

もう1つは、REGO.DATというテキストファイルだ。このファイルのなかには思いつくだけ多くの正規表現をひとつ1行でどんどん書いておく。少なくとも10個くらいは欲しいところだ（正規表現を勉強するためのモノだと言っておきながら、いきなりそれ自体を用意させてしまうのか…うーん）。たとえばこんな感じだ（**リスト4-2**）。ちなみに、行頭が"#"で始まっている行は注釈として無視される。

リスト4-2　<rego.dat>

```
# duumy rego.dat
[0-9]+
[a-z]+[:;]
[0-9]+[:;][[0-9]*
```

さて、以上の2つが揃ったところでようやくゲームが始められる（もちろん、後に載っているrego.awkも用意しておかなければならない）。読者諸氏はこの難関を乗り越えられただろうか？ ゲームを始めるには本スクリプトのデータとして先に用意した"REGO.DAT"を喰わせればよい（そう正規表現の書かれたファイルは、別に"REGO.DAT"でなくても良かったのだ）。

```
A>jgawk -f rego.awk rego.dat
```

スクリプトが実行されると画面がクリアされ、**図4-11**のような画面になるはずだ。ゲームの目的は、画面左上に表示される正規表現（これが問題なワケ）に一致する文字列を画面中央に表示されている文字列のなかから探し出すというものだ。画面中上に表示されたノルマの回数だけ一致する文字列を探し出せたら「あがり」となる。

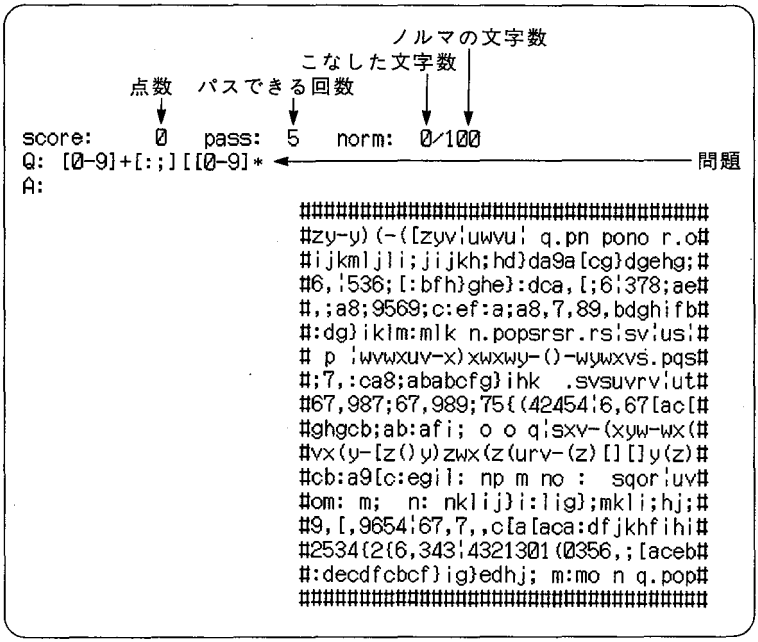


図 4–11 <rego.awk>実行例

各種キー操作は**表 4–1**に示すとおりだ。つまり 2、4、6、8 キーでカーソルを移動させて問題に一致する文字列の先頭で X キーを押してマークを開始、再度 2、4、6、8 でカーソルを移動させ、問題に一致する文字列の終わりで X キーを押してマークを終了させる。マークされた文字は反転表示される。マークする文字列は横だけでなく縦方向にも横に移動してから縦に移動するといった形でも指定できる。

キー操作	意　味
2,4,6,8	– カーソル移動 (下、左、右、上)
x	– マーク開始、終了
z	– マーク中断
q	– ゲーム中断
p	– パス (次の問題)
ESC	– パニック
CTRL+L	– 再表示

表 4–1　<rego.awk>のキー操作

では、図 4–11を解いてみよう。“[0–9]+[:;][[0–9]*” が出題されている。そこで中央付近へカーソルを移動して X キーで始点をマーク。次にカーソルを下に 5 つ、右へ 2 つ移動させ、X キーで今度は終点をマークした。

おや?

```
score:     0   pass:  5   norm:  0/100
Q: [0-9]+[:;][[0-9]*
A: uhgfnxwx
can't match!
```

図 4–12　<rego.awk>実行例

マークした部分が問題の正規表現と一致した場合には、マークした文字

は消去される。もし一致しなかった場合、マークした文字は赤の反転になりその部分に関しては二度とマークできない障害物として

あなたを邪魔してくる

つまりあまり適当にマークしていると、マークできる場所がなくなってしまうわけだ。

指定された問題の正規表現に一致する文字列がどうしても見つからない（本スクリプトは、あまり多くのことを考えて出題していないので、実際にない場合もある）場合は、“p”を入力することで、その問題をパスして次の問題に進むことができる。

本スクリプトで遊んでいる時に、

見られたくない人がやってきた

場合は、ESC キーを押すことによって画面を切り替えることができる。ESC キーを押した時に画面に表示されるのは、“panic”というファイルなので、このファイルにそれらしい内容（例えば MS-DOS のコマンドラインから C のコンパイルをかけている最中の画面など）を用意しておくとよい。

ノルマを達成するか、「もうわかんない」とか「あきた」とか「ほんとに面白いのかこれは！」とかいった理由で“q”キーを押すと、今までのハイスコアのベスト 10 を表示しスクリプトを終了する。この時直前に獲得した得点がベスト 10 に入っていた場合は名前を入力することができる。

解　説

この rego.awk は awk に不向きな会話的処理を駆使したプログラムである。メインルーチンは END のなかから呼び出され、普通のパターン、アクションは rego.dat を読み取る作業しかしていない。

対話的な部分は関数 select_str とその下位のルーチンにある。実際にキーを入力する部分は、awk だけではどうしようもないので system で外部コ

マンド batkey を呼び出している (関数 inkey)。古いバージョンの j?gawk だと動かないのもある (system がエラーコードを返さない) ので注意してもらいたい。

select_str 付近で一番トリッキーなのは、連想配列の添え字として選ばれた文字などの座標を入れていることだろう。sel がその配列である。

select_str では、すでに選ばれた文字かどうか判定する必要がでてくる。しかしループでぶん回して調べていたのでは遅すぎる。そこで awk にもともと備わっている、連想配列の添え字を検索する機能を利用したのだ。

しかし配列 sel に添え字 (x, y) があるかどうかを調べるのに if(sel[x, y]) {foo} という操作を行なうだけで、配列 sel が変化してしまい sel[x, y] という要素が出来てしまうので駄目だ。このケースでは if((x SUBSEP y) in sel) {foo} とすべきである。

問題点

編集担当 O のたっての希望で作られたゲームであるが、当の本人に明確なイメージがなく、えらいあやふやな仕様でスタートしてしまったため、ゲームデザインが単調でハマるほど面白くない。

ありがちー

たとえば文字の代わりに記号や色を使った正規表現にすると、よりパズルらしくなりもうすこし楽しめるかもしれない。またせっかく BATKEY.COM を利用してリアルタイムでキーを取得しているのだから、答えるまでの時間を計測して得点に加味すると緊迫感が出るかもしれない。

最初に文字列を作成する際に、何も考えず結構ランダムに出しているので問題によっては一致しないケースがある。ランダムな文字でなく、ある程度意味を持った文字を表示した方が正規表現の勉強にはよいだろう。

またここでは割愛したが、他の awk のスクリプトを読み込んで正規表現の

問題を自動作成する

スクリプトも簡単に用意できるので加えてみてはどうだろう。

スクリプト

REGO.AWK

```
#! gawk
# rego.awk
# regular expression is greek and obfuscated.
# S,Takeshi

# global
# quiz_num - 問題の数
# quiz_tab[] - 問題の配列
# quiz_str - 現在の問題の番号
# quiz_re - 現在の問題
# board[] - 盤面(固定長文字列の配列)
# block_num - 取ったblockの数
# pass_num - pass できる回数
# score - score
# cur_x, cur_y - 現在のcursor座標 (1origin)
# sel[] - board 上の選ばれた文字の座標
# sel_str - 選ばれた文字列
# sel_num - 選ばれた文字の数
# dead[] - 選べない文字の座標
# dead_num - 選べない文字の数

# constant
# block_clear - clear に必要なblockの数
# board_x, board_y - 盤面の大きさ
# org_x, org_y - 盤面の左上隅の画面上の座標
# rank_file - ranking file
# max_rank - 何位まで記録するか
# panic_file - panic時に表示するファイル
# pass_max - game開始時にできるpassの数

BEGIN {
    quiz_num = 0;
    board_x = 32; board_y = 15;
    block_clear = 100;
```

```
    org_x = (80 - board_x) / 2;  org_y = 4;
    pass_max = 5;
    srand();
    rank_file = "rego.scr";
    panic_file = "panic";
    max_rank = 10;
    dead_num = 0;
}

function cls()
{
    printf("¥033[2J");
}

function locate(x, y)
{
    printf("¥033[%s;%sH", y, x);
}

function xy2x(xy,   a)
{
    split(xy, a, SUBSEP);
    return a[1];
}

function xy2y(xy,   a)
{
    split(xy, a, SUBSEP);
    return a[2];
}

function inkey(     k)
{
    k = system("batkey 2 > nul");
    return k;
}

function make_board(        x, y, i, rands, rands_num)
{
    rands = "(012{34|56,789;,[abc:defg}hij;klm: nopq .rst|uvwx-(yz)[]";
    # 文字列randsを変えれば各記号の出現頻度を変えられる。
    rands_num = length(rands);

    for (y = 1; y <= board_y; y++) {
        i = int(rand() * rands_num) + 1;
        board[y] = "";
        for (x = 1; x <= board_x; x++) {
```

```
            do {
                r = rand() - .5;
                r = int(r * r * r * 48 + .5);
            } while (r == 0 || i + r < 1 || i + r > rands_num);
            i += r;
            board[y] = board[y] substr(rands, i, 1);
        }
    }
}

function show_score()
{
    printf("¥033[36m¥033[Hscore: %5d   pass: %2d   norm: %2d/%2d¥n",
     score, pass_num, block_num, block_clear);
    printf("Q: %s¥033[K¥n", quiz_str);
}

function show_answer()
{
    printf "¥033[36m¥033[3;1HA: %s¥033[K¥n", sel_str;
    printf "                    "; # erase message
}

function show_screen(   y)
{
    cls();
    show_score();
    show_answer();
    locate(org_x, org_y);
    printf "¥033[33m";
    for (x = 0; x <= board_x + 1; x++) printf "#";
    for (y = 1; y <= board_y; y++) {
        locate(org_x, org_y + y);
        printf "¥033[33m#¥033[0m%s¥033[33m#¥033[0m", board[y];
    }
    locate(org_x, org_y + board_y + 1);
    printf "¥033[33m";
    for (x = 0; x <= board_x + 1; x++) printf("#");
    highlight(dead, 41);
    printf "¥033[0m";
}

function highlight(ss, flag,        xy, c, x, y)
{
# {un,}highlight selected char
    for (xy in ss) {
        x = xy2x(xy); y = xy2y(xy);
```

```
        c = substr(board[y], x, 1);
        locate(org_x + x, org_y + y);
        printf "¥033[%sm%s¥033[0m", flag, c;
    }
}

function highlight_char(x, y, flag,     c)
{
    c = substr(board[y], x, 1);
    locate(org_x + x, org_y + y);
    printf "¥033[%sm%s¥033[0m", flag, c;
}

function panic()
{
    system("type " panic_file);
    inkey();
    show_screen();
}

function select_str(        c, x, y, mark)
{
    sel_num = 0; mark = 0;
    while (1) {
        locate(org_x + cur_x, org_y + cur_y);
        c = inkey();
        # print "¥033[Hbatkey " c;
        # 2468 - move
        # x(120) - mark start, end
        # z(122) - cancel
        # p(112) - pass
        # q(113) - quit
        # esc(27) - panic
        # ^L(12) - redraw
        x = cur_x; y = cur_y;
        if      (c == 50 && y < board_y) y++;
        else if (c == 52 && x > 1) x--;
        else if (c == 54 && x < board_x) x++;
        else if (c == 56 && y > 1) y--;
        else if (c == 120) {
            mark = !mark;
            if (mark) {
                if ((x SUBSEP y) in dead) {
                    mark = 0;
                } else {
                    sel_num = 1;
                    sel[cur_x, cur_y] = 1;
```

```
                sel_str = substr(board[cur_y], cur_x, 1);
                highlight_char(x, y, 7);
                show_answer();
            }
        } else {
            return "";
        }
    } else if (c == 122) {
        mark = 0;
        for (xy in sel) {
            highlight_char(xy2x(xy), xy2y(xy), 0);
            delete sel[xy];
        }
    } else if (c == 112) {
        if (!mark) return "pass";
    } else if (c == 113) {
        return "quit";
    } else if (c == 12) {
        show_screen();
        highlight(sel, 7);
    } else if (c == 27) {
        panic();
    }
    # check
    if (mark && (x != cur_x || y != cur_y)) {
        if ((x SUBSEP y) in sel && sel[x, y] == sel_num - 1) {
            sel_num--;
            highlight_char(cur_x, cur_y, 0);
            delete sel[cur_x, cur_y];
            sel_str = substr(sel_str, 1, sel_num);
            show_answer();
        } else if ((x SUBSEP y) in sel || (x SUBSEP y) in dead) {
            x = cur_x; y = cur_y;
        } else {
            sel_num++;
            sel[x, y] = sel_num;
            sel_str = sel_str substr(board[y], x, 1);
            highlight_char(x, y, 7);
            show_answer();
        }
    }
    cur_x = x; cur_y = y;
  }
}

function update_board(      xy, x, y)
{
```

```
    for (xy in sel) {
        x = xy2x(xy); y = xy2y(xy);
        locate(org_x + x, org_y + y); print " ";
        board[y] = substr(board[y],1,x-1) " " substr(board[y],x+1);
        delete sel[xy];
    }
}

function ranking(       i, scores, names, rank)
{
    rank = 0;
    for (i = 1; (getline < rank_file) > 0; i++) {
        if (rank == 0 && $2 <= score) {
            scores[i] = score;
            rank = i;
            i++;
        }
        scores[i] = $2; names[i] = $3;
    }
    close(rank_file);
    if (i <= 1) { # is rank file empty?
        scores[1] = score; rank = 1;
    }

    printf "¥033[0m";
    for (i = 1; i <= max_rank; i++) {
        if (i == rank) {
            printf "¥033[7m%2d %9d %s¥033[0m¥n",
             i, scores[i], names[i];
        } else {
            printf "%2d %9d %s¥n", i, scores[i], names[i];
        }
    }

    if (rank) {
        printf("you are rank %d. enter your name:", rank);
        getline  < "con"; names[rank] = $1;
        printf "" > rank_file; # trucate
        for (i = 1; i <= max_rank; i++) {
            printf "%2d %9d %s¥n",
             i, scores[i], names[i] >> rank_file;
        }
        close(rank_file);
    }
}

function quit()
```

```
{
    cls();
    ranking();
    exit;
}

function sel_quiz(      i, j)
{
    i = int(rand() * 4) + 1;
    quiz_str = "";
    while (i--) {
        quiz_str = quiz_str quiz_tab[int(rand() * quiz_num)];
        quiz_str = quiz_str substr("*+?", rand() * 5, 1);
    }
    quiz_re = "^" quiz_str "$";
}

function play(      s)
{
    pass_num = pass_max;
    cur_x = cur_y = 1;
    make_board();
    show_screen();
    while (block_num < block_clear) {
        # 問題を出す (pass も出来るようにしよう)
        sel_quiz();
        show_score();
        while (1) {
            s = select_str();
            if (s == "quit") quit();
            if (s == "pass") {
                if(pass_num > 0) {
                    pass_num--;
                    break;
                }
                printf "¥033[4;1H¥033[31mno more pass!";
                continue;
            }
            if (sel_str ~ quiz_re) {
                score += sel_num * sel_num;
                block_num += sel_num;
                update_board();
                printf "¥033[4;1H¥033[33mmatch!";
                break;
            }
            printf "¥a¥033[4;1H¥033[31mcan't match!¥a";
            highlight(sel, 41);
```

```
            for (xy in sel) {
                dead[xy] = 1;
                delete sel[xy];
            }
            dead_num += sel_num;
        }
        show_score();
    }
    # clear の処理
    block_num -= block_clear;
    score += block_num * block_num * block_num;
    quit();
}

/^[^#]/ {
    quiz_tab[quiz_num] = $0;
    quiz_num++;
}

END {
    play();
}
```

REGO.DAT

次に示すのは今回使用した出題ファイルの内容である。awk ではこれらの問題自体を自動生成（検索して抽出すればよい）することも十分可能である。

```
# dummy rego.dat
[0-9]+
[a-z]+[:;]
[0-9]+[:;][[0-9]*
```

運勢を強くしろ!

姓名バリバリチューン!!

機　能

姓名判断をもとに画数を変化させることで、自分の理想とする姓名運に近づいた新しい名前を提案する。よりよい運勢の名前を探求するという画期的な（ほんとかー!）姓名最適化スクリプトだ。

操　作

本スクリプトを実行する前に、運勢を占うためのアンチョコである、“point.tab”というテキストファイルを用意する。これを間違えると全員の運勢がめちゃくちゃになる。

“point.tab”ができあがったら、以下のように本スクリプトに「姓」と「名」をスペースで区切って指定する。すると名前に使用されている

漢字の画数を聞いてくる

ので、空に字を書きながら画数を数えて入力してやる。ここで入力した文字は“kakusu.dic”というファイルに格納され、次からその字はここで指定した画数が使用されるので慎重に行なうこと。間違って入力すると提案される名前まででたらめになる。

```
A>jgawk -f naming.awk 志村　拓
「志」の画数を教えてください:
```

図4–13 naming.awk の実行 1

“kakusu.dic” にない文字の画数をすべて入力し終わると、現在の名前の姓名判断が表示される。それぞれの運は 100 点制で表示され

点数が大きい方が運がイイ

ということになるのだ。

現在の名前の姓名判断が表示された後に、本スクリプトのウリである、“姓名バリバリチューン” のための、

どの運勢をチューンしますか？(1-5):

の質問が表示される。ここでチューンアップしたい運（つまり点数の低い運）を、その前に表示されている番号で指定する。

```
A>jgawk -f naming.awk 志村　拓
「志」の画数を教えてください: 7
「拓」の画数を教えてください: 8

あなたの名前は　志　村　拓
　　画数は　 7　 7　 8
1.天運: 14画 80点　　2.地運:  8画 80点
3.人運: 15画 10点　　4.外運: 15画 10点　　5.総運: 22画 80点

どの運勢をチューンしますか？(1-5):
```

図4–14 naming.awk の実行 2

すると指定した運の点数が最も高くなる名前を、

厳正なる審査の後

適当にでっち上げて作成し、さらにその名前に変更した時の運勢を表示する。

```
A>jgawk -f naming.awk 志村　拓
「志」の画数を教えてください: 7
「拓」の画数を教えてください: 8

あなたの名前は　志　村　拓
　　画数は　 7　 7　 8
1.天運: 14画 80点　　2.地運:  8画 80点
3.人運: 15画 10点　　4.外運: 15画 10点　　5.総運: 22画 80点

どの運勢をチューンしますか？(1-5): 3

氏名の3番目を18画にチューンします

志村麿
志村曙
志村鞭
志村癖
志村難

チューンの成果は
1.天運: 14画 80点　　2.地運: 18画 80点
3.人運: 25画100点　　4.外運: 25画100点　　5.総運: 32画100点

A>
```

図4-15 naming.awkの実行2

解　説

ひと昔前に、巷にフリーで流れていた姓名判断のプログラムを妊婦の妻を持つ知人にコピーしたところ、そのソフトは瞬く間にさる

産婦人科の病院で大ヒットソフト

となった。そう、これから産まれる我が子が少しでもいい運勢に恵まれるようにという藁にもすがる思いの親心が、この「なんの根拠もない（といっても、このソフト、ちゃんと占いの本を見て組まれていたようだが）」姓名判断ソフトを大流行させてしまったわけだ。

「決定している名前を占うにはこれで充分なんだケド、これから名前を決めるには、この仕様ではキツイんだよね。名前を決めては占い直す Try & Go の繰り返しだから」というのが当時の友人の弁である。それでは、ということで作成したのが、今回の「姓名バリバリチューン」プログラムである。ちなみにその時、試行錯誤のあげく最高得点を獲得した史上最強の名前は、

魔人我亜　乙!

だったそうである。

問題点

まずあやしいのは、そもそも姓名判断がこういう使われ方を想定して作られていないことだ。今回は一般的とされる評価をもとに数値化してもらったが、運勢を一元的に数値化すること自体に無理があるし（いばるんじゃない）、その点数評価のレベル数も本プログラムでは少な過ぎる。さらにその数値を最適化するように演算するなんてことをやった人などほとんどいないだろう。だいたい、

生まれたばかりの赤ちゃん

を除けば、名前なんてそんな簡単に変えられる訳じゃないんだから。

このプログラムによって名前が決定してしまった未来を担う人へ（いないと思うけど）、幸運な人生が送れるよう蔭ながらお祈りします。

スクリプト

NAMING.AWK

```
#
#  naming.awk - 姓名最適化スクリプト
#
#  Copyright (C) 1993 by SASAKI Takeshi        協力: keiz
#
#  説明
#    姓名の画数から運勢を判定し、より運のよい画数の文字を提案する。
#
#    データベースファイル kakusu.dic をカレントディレクト
#    リに作成する。文字の画数はこのファイルから得る。与え
#    られた文字が kakusu.dic にない場合は、画数を要求して
#    くるので答えること。以後、その文字は kakusu.dic に登
#    録され、同じ文字について二度尋ねることはない。
#

BEGIN {
    if (ARGC != 3) {
        print "usage: jgawk -f seimei.awk <姓> <名>" > "/dev/stderr"
        exit(1)
    }

    kakudic = "kakusu.dic";
    untab = "point.tab";

    sei = ARGV[1]; mei = ARGV[2];
    simei = sei mei;
    slen = jlength(sei); mlen = jlength(mei);
    len = slen + mlen;
    ARGV[1] = ARGV[2] = ""

    make_map();
    read_point();
    read_kakudic();

    if (getkakusu(simei, kaku, len)) {
        addkakusu(simei, kaku, len);
    }
    print;

    printf "あなたの名前は¥t";
    for (i = 1; i <= len; i++)
```

```
        printf "%s ", jsubstr(simei, i, 1);
    printf "¥n";
    printf "¥t画数は¥t";
    for (i = 1; i <= len; i++)
        printf "%2d ", kaku[i];
    printf "¥n";

    calc_all();

    printf "¥nどの運勢をチューンしますか？(1-5): "; getline k;
    print;
    if (k == 1)      tuneup(ten, ten_map);
    else if (k == 2) tuneup(ti,  ti_map);
    else if (k == 3) tuneup(jin, jin_map);
    else if (k == 4) tuneup(gai, gai_map);
    else if (k == 5) tuneup(sou, sou_map);


}

function read_kakudic()
{
    while (getline < kakudic) {
        kaku_tab[$2] = kaku_tab[$2] $1;
    }
}

function read_point(    i)
{
    while (getline < untab) {
        for (i = 2; i <= NF; i++) {
            point[$i] = $1;
            if ($i > kaku_max) {
                kaku_max = $i;
            }
        }
    }
}

function getkakusu(s, k, n,     i, j, c, r, rr)
{
    r = 0;
    for (j = 1; j <= n; j++) {
        c = jsubstr(s, j, 1);
        rr = 1;
        for (i in kaku_tab) {
            if (jindex(kaku_tab[i], c)) {
```

```
                k[j] = i; rr = 0;
                break;
            }
        }
        r = r || rr;
    }
    return r;
}

function addkakusu(s, k, n,     i)
{
    for (i = 1; i <= n; i++) {
        if (k[i] == 0) {
            printf("「%s」の画数を教えてください: ", jsubstr(s, i, 1));
            getline k[i];
            print jsubstr(s, i, 1), k[i] >>kakudic;
        }
    }
}

function make_map(      i)
{
    for (i = 1; i <= slen; i++) {
        ten_map[i] = sou_map[i] = gai_map[i] = 1;
    }
    for (i = 1; i <= mlen; i++) {
        ti_map[slen + i] = sou_map[slen + i] = gai_map[slen + i] = 1;
    }
    jin_map[slen] = jin_map[slen + 1] = 1;
    if (slen > 1) delete gai_map[slen];
    if (mlen > 1) delete gai_map[slen + 1];
}

function calc_all()
{
    ten = calc_point(ten_map);
    ti  = calc_point(ti_map);
    jin = calc_point(jin_map);
    gai = calc_point(gai_map);
    sou = calc_point(sou_map);
    printf("1.天運: %2d画%4d点¥t", ten, point[ten] * 100);
    printf("2.地運: %2d画%4d点¥n", ti, point[ti] * 100);
    printf("3.人運: %2d画%4d点¥t", jin, point[jin] * 100);
    printf("4.外運: %2d画%4d点¥t", gai, point[gai] * 100);
    printf("5.総運: %2d画%4d点¥n", sou, point[sou] * 100);
}
```

```
function calc_point(map,        i, n)
{
    n = 0;
    for (i in map) n += kaku[i];
    return n;
}

function tuneup(un, map,        i, j, n, kouho, kaku_kouho, point_
max, new)
{
    point_max = point[un]; j = 0;
    for (i = 1; i < kaku_max; i++) {
        if (point[i] > point_max) {
            kaku_kouho[1] = i; j = 2; point_max = point[i];
        } else if (point[i] == point_max) {
            kaku_kouho[j++] = i;
        }
    }
    if (point_max == point[un]) {
        print "これ以上よくなりません";
        return -1;
    }
    # どの文字を増やす?
    for (n = len; n >= 1; n--) if (n in map) break;
    for (i = 1; i < j &&  0 > kaku[n] + kaku_kouho[i] - un; i++);
    kaku[n] += kaku_kouho[i] - un;

    printf "氏名の%d番目を%d画にチューンします¥n",n,kaku[n];
    print;

    if("" != (kouho = kaku_tab[kaku[n]])) {
        new = ""
        for (i = 1; i <= jlength(kouho); i++) {
            new = jsubstr(simei,1,n-1) jsubstr(kouho,i,1) jsubstr
(simei,n+1);
            print new;
        }
    } else {
        print "そのような文字は知りません";
    }
    print;

    print "チューンの成果は";
    calc_all();

    return 0;
}
```

POINT.TAB

以下に示すのは画数から導きだされる運勢を評価し、数値に落とし込んだデータをテーブルにまとめたものである。

```
0.05    29 57
0.10    11 35 55 70
0.20    3 5 10 13 15 21 23 27 37 41 45 56 60 64 66 69 74 76 79 80
0.40    20 26 28 30 31 43 44 47 50 51 52 54 62 71 72 77 78
0.80    2 6 7 8 9 12 14 16 17 18 19 22 24 33 36 38 39 42 46 48
0.90    4 49 58 59 61 67 68 73 75
1.00    1 25 32 34 40 53 63 65 81
```

図 4–16 point.tab

KAKUSU.DIC

今回のスクリプトでは画数がわからない場合（画数辞書に未入力の場合）、入力者に問い返すようになっている。ただ、名前をチューンする時に提案される文字の種類が少ないとまるでつまらないので画数辞書は豊富な方がよい。次に今回使用したデータファイルの一部を示すが、当然ながら変な“名前”が提示された方が、

絶対!!おもしろい

のだから、わざと変な文字をこの“kakusu.dic”に入力しておいた方が場が盛り上がるのは言うまでもない。以下に検討の結果求められた「お勧め辞書」の内容を示す。まあ、このあたりは必須と言ってよいだろう。

鬼 10
魔 21
我 7
乙 1
猫 11
凸 5
凹 5
平 5
六 4
牛 4
卍 6
吉 6
寅 11
足 7
漬 14
物 8
馬 10
釜 10
岩 8
樽 16
丸 3
井 5
熊 14
茶 9
八 2
桃 10
為 9

図 4–17 画数辞書の一部

ひとり一台? えっ大勢一台!! じゃ必要です!

ユーザー管理 awk

機 能

ユーザー管理 awk は、以下の 4 つのスクリプト群から構成される。

ユーザー管理AWK

LOGIN.AWK	ユーザーの使用開始を管理し環境を整える
LOGOUT.AWK	ユーザーの使用終了時に使用時間を記憶する
PASSWD.AWK	"/etc/passwd"のパスワードを管理する
LAST.AWK	各ユーザーのマシン占有状況を報告する

これらは UNIX の login(1) をまねたユーザーロギングを行うためのものだ。各スクリプトの名称は UNIX のコマンドに準拠している。また、各スクリプトが使用するいくつかのデータファイルのパス名も UNIX のそれにならっている。

本スクリプト群を使用することにより、誰がどのくらいコンピュータを占有したかの記録を取るとともに、パスワードを指定しておくことによってセキュリティを（ほんのわずか？）高めることができる。また、UNIX の雰囲気をパソコンで味わうことや、UNIX の古典的ハッキング手段である「トロイの木馬」のベースとしても使用することができる（ヲイヲイ）。

操　作

本システムを使用する前に、図4–19に示すいくつか用意しておかなければならないファイルがある。ここでは紙面の都合から（オレにもっと紙をよこせぇぇ^_^）、これらのファイルの例を示すとともに、ごく簡単な説明をするだけにとどめる。基本的にはBSD系UNIXのファイルとコンパチになっているので、詳しくはUNIXの参考書やオンラインマニュアルを参照して欲しい。

ちなみに、"/etc/gettytabs"では、"im"のエントリ（gettyのプロンプト文字列）に"%v"（OSのバージョン）、"%h"（ホスト名）、"%t"（tty名）が使用できる。また、"/etc/passwd"のパスワードの部分は、UNIXのcrypt(3)との互換性はなく、独自の暗号化（というにはおこがましいが…）ロジックを使用しているので、UNIXの"/etc/passwd"をそのまま流用することはできないので注意してほしい。

用意するファイル

/etc/gettytab	ターミナル設定のテーブル
/etc/motd	ログインした後に表示されるメッセージ
/etc/passwd	ユーザー情報のテーブル
/etc/ttys	各ターミナルの設定情報

上記のファイルを作成し環境整備を行うとようやく本システムのスクリプト群が使用可能となる。本システムは、UNIXのユーザー承認システムに関連するいくつかのコマンドをawkのスクリプトで再現したものである。以下に本システムの各スクリプトの使い方を説明する。

LOGIN.AWK <コンピュータ使用開始を宣言>

まず、本スクリプトを"jgawk –f login.awk"と入力して実行すると、"/etc/ttys"から、gettyの使用するタイプを取得し、そのタイプを"/etc

/gettytab" から探しそのなかの getty のプロンプト文字列を取得する。このプロンプトと

```
Login:
```

という文字を表示し、ユーザーに ID の入力を促す。ここでユーザーが ID を入力すると、次に

```
Passwd:
```

という文字を表示し、パスワードの入力をユーザーに促す（ただし、指定された ID にパスワードが設定されていなかった場合は、このプロンプトはスキップされる）。

入力された ID が登録されているもので、パスワードが妥当であるか否かを、"/etc/passwd" ファイルを見て調べ、もし ID が登録されていなかったり、パスワードが妥当でなかった場合、再度 "Login: " という文字が表示され、ID を入力するところからやり直しとなる。

入力された ID とパスワードが妥当であった場合は、"/usr/adm/wtmp" に、今回のログインの状況を記録するとともに、同ファイルから最後にログインした時間を画面上に表示する。最後に "/etc/passwd" に設定されている各ユーザーのホームディレクトリにカレントディレクトリを移して、スクリプトを終了する。

LOGOUT.AWK <コンピュータ使用終了を宣言>

"jgawk –f logout.awk" と入力し本スクリプトを実行すると、現在の時刻を "/usr/adm/wtmp" に書き込む。

PASSWD.AWK <ユーザーのパスワードを変更>

"jgawk –f passwd.awk" と入力し本スクリプトを実行すると、

```
Login name:
```

と表示され、パスワードを変更するユーザーの ID の入力を促す。ここで ID を入力すると、すでにそのユーザーにパスワードが設定されている場合は、

```
Old password:
```

と表示され、現在設定されているパスワードの入力を促す。ここで間違ったパスワードを入力すると本スクリプトは中断終了される。次に、

```
New password:
```

と表示されるので新しく設定するパスワードを入力する。最後に、

```
Retype new password:
```

と表示されるので、確認のために再度新しく設定するパスワードを入力する。ここで入力した 2 つの新しく設定するパスワードが同じ場合に限り、"/etc/passwd" ファイルに新しいパスワードが設定される。

LAST.AWK <各ユーザーのマシン占有状況を報告>

"jgawk -f last.awk" と入力し本スクリプトを実行すると、現在までの各ユーザーのマシン占有状況を "/usr/adm/wtmp" から出力する。もちろん前述の LOGIN.AWK 及び LOGOUT.AWK を使って記録されたものだ。また

```
jgawk -f last.awk <ユーザー名>
```

といった具合にパラメータとしてユーザー名を指定すると、そのユーザーだけのマシン占有状況が出力される。

解　説

その他のスクリプトとしてCRYPT.AWKとDATE.AWKが、本パッケージ内には同包されている。

CRYPT.AWKは、LOGIN.AWK及びPASSWD.AWKで使用している暗号化スクリプトを抜き出したものである。

ユーザーのうちで、"/etc/passwd"で指定されているパスワードのチェックを行うスクリプトを作成する場合は、本スクリプトをたたき台にして、入力されたパスワードを本スクリプトのCrypt()関数にとおして"/etc/passwd"の暗号化されたパスワードと比較し、同じであればパスワードが一致したとみなすようにする。

DATE.AWKはsrand()システム関数から、現在の年月日時分及び曜日を取得するものだ。このスクリプトは、引数として1970年1月1日からの経過秒数を指定することでその値を年月日時分及び曜日に変換するといった機能もある。この機能を利用すれば、"/usr/adm/wtmp"に記録されているログイン時間、ログアウト時間を年月日時分及び曜日に変換することができる。

問題点

LOGIN.AWKは本来なら、"/etc/passwd"の起動シェルを子プロセスとして実行したいところだが、そこまでやってしまうとコンベンショナルメモリも狭くなるし…ということでとりやめた。

通常UNIXでは、ログアウトすると制御がその元であるinit(8)に戻り、getty(8)が再度fork & execされるが、logout.awkではそのままcommand.comに制御が戻ってしまう。以下のようなバッチファイルを用意して、本スクリプト終了後にLOGIN.AWKを実行するようにすべきだろう。

リスト 4-3　スクリプト終了後に LOGIN.AWK を実行する

```
echo off
jgawk -f LOGOUT.AWK
jgawk -f LOGIN.AWK
```

最初の“Login name: ”は、UNIX の passwd(1) にはないフェーズであるが、現在使用中のユーザー ID を取得する手段がないのでこうした不細工なものになってしまった。

冒頭でも説明したとおり、passwd.awk や login.awk の中で使用しているパスワード暗号化ロジックに UNIX の crypt(3) との互換性はない。

LOGIN.AWK で“/usr/spool/mail”ディレクトリの下に自分宛のメールがスプールされていた場合、“You have [new] mail.”のメッセージを表示するようにするとともに、ucbmail とまではいかずとも、binmail の真似事をする mail(1) を模倣するスクリプトを用意するとさらに楽しいかもしれない。

“/usr/adm/wtmp”があるのであれば、UNIX の ac(1) のようなログの統計を行うようなスクリプトを用意したいところだ。

スクリプト

LOGIN.AWK

```
#
# LOGIN
# Copyright (c) 1993 by Taku-SHIMURA
#

BEGIN {
    # Init. variables
    True      = (1 == 1)
    False     = (1 == 0)
    EchoOn    = "¥033[0m"
    EchoOff   = "¥033[8m"
    OsVersion = GetVersion()
    HostName  = GetHostname()
    TtyName   = "console"

    # "/etc/getty" phase
    prompt    = GetGettyType(TtyName, OsVersion, HostName)
    for(;;) {
        do {
            printf prompt "Login: "
            getline name < "/dev/stdin"
        } while(0 == length(name))

        # "/bin/login" phase
        if(Login(name)) {
            exit
        }
        Sleep(10)
    }
}

END {
    # logging to "/usr/adm/wtmp"
    logfile = "wtmp"
    newlog  = logfile ".new"
    GetDate()
    printf "%s %s %10d¥n", PwLogin, TtyName, Time > newlog
    while(getline < logfile > 0) {
        print $0 >> newlog
        if("" == LastLog && $1 == PwLogin) {
            print "Last login: " GetDate($3 + 0) " on " $2
```

```
            LastLog = 1
        }
    }
    close(newlog)
    system("del " logfile)
    system("ren " newlog " " logfile)

    # print "/etc/motd"
    motd = "motd"
    while(getline < motd > 0) {
        print $0
    }
}


# "/bin/login"
func Login(name)
{
    for(loop = 0; loop < 5; ++loop) {
        while(0 == length(name)) {
            printf "login: "
            getline name < "/dev/stdin"
        }
        findID =  GetPasswd(name)
        if(findID && 0 == length(PwPasswd)) {
            return True
        }
        printf "Passwd: " EchoOff
        getline passwd < "/dev/stdin"
        printf EchoOn
        passwd = Crypt(passwd)
        if(findID && PwPasswd == passwd) {
            return True
        }
        printf "Login incorrect¥n"
        name = ""
    }
    return False
}


# Get OS version by "ver" command.
func GetVersion(            tmpRet)
{
    while ( "ver" | getline > 0) {
        if(0 < NF) {
            tmpRet = $3
```

```
        }
    }
    if("" == OsVersion) {
        tmpRet = "4.2"
    }
    return tmpRet ""
}


# Get host name from "HOSTNAME" environment variable
func GetHostname(           tmpRet)
{
    if ("HOSTNAME" in ENVIRON) {
        tmpRet = ENVIRON["HOSTNAME"]
    } else {
        tmpRet = "myname"
    }
    return tmpRet
}


# Get login prompt from "/etc/ttys" & "/dev/gettytab"
func GetGettyType(tty, ver, host, tmpRet, tmpType, tmpLine, tmpRec,
tmpFld, tmpTag)
{
    # Research console tty type from "/etc/ttys"
    while (getline < "ttys" > 0) {
        if (4 < NF && tty == $1) {
            close("ttys")
            if("¥"" == substr($3, length($3))) {
                tmpType = substr($3, 1, length($3) - 1)
            } else {
                return ""
            }
            break
        }
    }

    # Research init getty prompt form "/etc/gettytab"
    while (getline tmpLine < "gettytab" > 0) {
        if(match(tmpLine, "[ ¥t]+$")) {
            tmpLine = substr(tmpLine, 1, RSTART)
        }
        if(match(tmpLine, "^#") || 0 == length(tmpLine)) {
            continue
        }
        tmpRec = tmpRec tmpLine
```

```
        if(":" == substr(tmpRec, length(tmpRec))) {
            split(tmpRec, tmpFld, ":")
            split(tmpFld[1],  tmpTag,   "|")
            for(tmpTag in tmpTag) {
                tmpTag = tmpTag[tmpTag]
                if(tmpType == tmpTag || "default" == tmpTag) {
                    for(tmpFld in tmpFld) {
                        tmpFld = tmpFld[tmpFld]
                        if("im=" == substr(tmpFld, 1, 3)) {
                            tmpRet = substr(tmpFld, 4)
                        }
                        break;
                    }
                }
            }
            if(tmpType == tmpTag && "" != tmpRet) {
                break
            }
            tmpRec = ""
        }
    }
    gsub(/%h/,    host,   tmpRet)
    gsub(/%v/,    ver,    tmpRet)
    gsub(/%t/,    tty,    tmpRet)
    gsub(/¥¥n/,   "¥n",   tmpRet)
    gsub(/¥¥r/,   "¥r",   tmpRet)
    gsub(/¥¥033/, "¥033", tmpRet)
    close("gettytab")

    return tmpRet
}


# Get pass word from "/etc/passwd"
func GetPasswd(name,        tmpRec, tmpFld)
{
    while (getline tmpRec < "passwd" > 0) {
        tmpFld = split(tmpRec, tmpFld, ":")
        if(7 == tmpFld && name == tmpFld[1]) {
            PwLogin  = tmpFld[1]
            PwPasswd = tmpFld[2]
            PwHome   = tmpFld[6]
            PwShell  = tmpFld[7]
            gsub(/¥//, "¥¥", PwHome)
            gsub(/¥//, "¥¥", PwShell)
            close("passwd")
            return True
```

```
        }
    }
    close("passwd")
    return False
}


# Sleep program
func Sleep(sec)
{
    srand()
    time = srand() + sec
    while(srand() < time) ;
}


# Crypt strings (Of course, this is mere shum.)
func Crypt(str,     tmpRet, tmpLop, tmpKey, tmpChr)
{
    if("" == _nCodeToChr) {
        _nCodeToChr = split("! ¥" # $ % & ' ( ) * + , - . / 0 1 2 3 4
 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V
 W X Y Z [ ¥¥ ] ^ _ a b c d e f g h i j k l m n o p q r s t u v w x
 y z { | } ~",
 _CodeToChr, " ")
        _CodeToChr[0] = " "
        for(tmpLop = 0; tmpLop <= _nCodeToChr; ++tmpLop) {
            _ChrToCode[_CodeToChr[tmpLop]] = tmpLop
        }
        CodeToChr[_ChrToCode[":"]] = "'"
    }
    tmpRet = ""
    tmpKey = _ChrToCode[substr(str, length(str))]
    for(tmpLop = 1; tmpLop <= length(str);  ++tmpLop) {
        tmpChr = _ChrToCode[substr(str, tmpLop,     1)]
        tmpKey = (tmpChr * (tmpKey + 1)) % _nCodeToChr
        tmpRet = tmpRet _CodeToChr[tmpKey]
    }
    return tmpRet
}


# GETDATE (Get today)
func GetDate(val,       leap) {
    if("" == _MonName[1]) {
        _TimeZone = 9 * 3600
        split("Sun Mon Tue Wed Thu Fri Sat", _WeekName, " ")
```

```
        split("Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec",
                        _MonName, " ")
        split("31 28 31 30 31 30 31 31 30 31 30 31 31 29 31 30
31 30 31 31 30 31 30 31", _MonLen, " ")
    }

    if(0 == val + 0) {
        srand()
        Time = srand() + _TimeZone
    } else {
        Time = val
    }

    Sec  = Time % 60
    Min  = int(Time / 60  ) % 60
    Hour = int(Time / 3600) % 24
    Day  = int(Time / (3600 * 24))
    Week = (Day + 4) % 7 + 1
    Year = 1970;
    for(++Day;;) {
        leap = ((Year%4)==0 && (Year%100)!=0 || (Year%400)==0) ? 12: 0
        yd   = (leap) ? 366: 365
        if(Day < yd)
            break
        ++Year
        Day -= yd
    }
    for(Mon = 1; _MonLen[Mon+leap]+0 <= Day+0; ++Mon) {
        Day -= _MonLen[Mon+leap];
    }
    return sprintf("%.3s %.3s%3d %02d:%02d:%02d JST %d",
        _WeekName[Week], _MonName[Mon], Day,
        Hour, Min, Sec, Year)
}
```

LOGOUT.AWK

```
BEGIN {
    # logging to "/usr/adm/wtmp"
    logfile = "wtmp"
    newlog  = logfile ".new"
    printf "" > newlog
    while(getline < logfile > 0) {
        printf $0 >> newlog
        if(0 == Time && 3 == NF) {
```

```
            GetDate()
            printf " %10d¥n", Time >> newlog
        } else {
            printf "¥n" >> newlog
        }
    }
    close(newlog)
    system("del " logfile)
    system("ren " newlog " " logfile)

}


# GETDATE (Get today)
func GetDate(val,       leap) {
    if("" == _MonName[1]) {
        _TimeZone = 9 * 3600
        split("Sun Mon Tue Wed Thu Fri Sat", _WeekName, " ")
        split("Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec",
                    _MonName, " ")
        split("31 28 31 30 31 30 31 31 30 31 30 31 31 29 31 30
                    31 30 31 31 30 31 30 31", _MonLen, " ")
    }

    if(0 == val + 0) {
        srand()
        Time = srand() + _TimeZone
    } else {
        Time = val
    }

    Sec  = Time % 60
    Min  = int(Time / 60 ) % 60
    Hour = int(Time / 3600) % 24
    Day  = int(Time / (3600 * 24))
    Week = (Day + 4) % 7 + 1
    Year = 1970;
    for(++Day;;) {
    leap = ((Year%4)==0 && (Year%100)!=0 || (Year%400)==0) ? 12: 0
    yd   = (leap) ? 366: 365
    if(Day < yd)
        break
    ++Year
    Day -= yd
    }
    for(Mon = 1; _MonLen[Mon+leap]+0 <= Day+0; ++Mon) {
        Day -= _MonLen[Mon+leap];
```

```
    }
    return sprintf("%.3s %.3s%3d %02d:%02d:%02d JST %d",
        _WeekName[Week], _MonName[Mon], Day,
        Hour, Min, Sec, Year)
}
```

PASSWD.AWK

```
#
# PASSWD (Change pass word in "/etc/passwd")
# Copyright (c) 1993 by Taku-SHIMURA
#

BEGIN {
    FS = OFS = ":"
    EchoOn   = "¥033[0m"
    EchoOff  = "¥033[8m"
    SrcFile  = "passwd"
    NewFile  = SrcFile ".new"
    ARGC     = 2
    ARGV[1]  = SrcFile

    Name = GetLoginName()
    if(0 == length(Name)) {
        exit
    }
    printf "" > NewFile
}


Name == $1 {
    if(length($2)) {
        printf "Old password: " EchoOff
        getline oldPasswd < "/dev/stdin"
        printf EchoOn
        oldPasswd = Crypt(oldPasswd)
        if($2 != oldPasswd) {
            printf "Sorry.¥n"
            next
        }
    }
    printf "New password: " EchoOff
    getline newPasswd < "/dev/stdin"
    printf EchoOn
    printf "Retype new password: " EchoOff
    getline retypePasswd < "/dev/stdin"
```

```
    printf EchoOn
    if(newPasswd != retypePasswd) {
        printf "Mismatch - password unchanged.¥n"
        exit
    }
    $2 = Passwd = Crypt(newPasswd)
}


{
    print $0 >> NewFile
}


END {
    close(NewFile)
    if(length(Passwd)) {
        system(sprintf("del %s", SrcFile))
        system(sprintf("ren %s %s", NewFile, SrcFile))
    } else {
        system(sprintf("del %s", NewFile))
    }
}


# Get login name
func GetLoginName(      tmpRet)
{
    # Properly speaking, use "USER" environment variable...
    # return ENVIRONMENT("USER")
    printf "Login name: "
    getline tmpRet < "/dev/stdin"
    return tmpRet
}


# Crypt strings (Of course, this is mere shum.)
func Crypt(str,     tmpRet, tmpLop, tmpKey, tmpChr)
{
    if("" == _nCodeToChr) {
        _nCodeToChr = split("! ¥" # $ % & ' ( ) * + , - . / 0 1 2 3 4
5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V
W X Y Z [ ¥¥ ] ^ _ a b c d e f g h i j k l m n o p q r s t u v w x
y z { | }
~",  _CodeToChr, " ")
        _CodeToChr[0] = " "
        for(tmpLop = 0; tmpLop <= _nCodeToChr; ++tmpLop) {
```

```
            _ChrToCode[_CodeToChr[tmpLop]] = tmpLop
        }
        CodeToChr[_ChrToCode[":"]] = "‘"
    }
    tmpRet = ""
    tmpKey = _ChrToCode[substr(str, length(str))]
    for(tmpLop = 1; tmpLop <= length(str);  ++tmpLop) {
        tmpChr = _ChrToCode[substr(str, tmpLop,     1)]
        tmpKey = (tmpChr * (tmpKey + 1)) % _nCodeToChr
        tmpRet = tmpRet _CodeToChr[tmpKey]
    }
    return tmpRet
}
```

LAST.AWK

```
BEGIN {
    ARGC    = 2
    ARGV[1] = "wtmp"
}

{
    GetDate($3)
    printf "%-8s  %-8s %16s %.3s %.3s%3d %02d:%02d",
      $1, $2, RHost, _WeekName[Week], _MonName[Mon], Day, Hour, Min
    if(3 < NF) {
        GetDate($4)
        printf " - %02d:%02d  (", Hour, Min
        GetDate($4 - $3)
        if(1 < Day) {
            printf "%d+", Day - 1
        }
        printf "%02d:%02d)¥n", Hour, Min
    } else {
        printf "   still logged in¥n"
    }
}


# GETDATE (Get today)
func GetDate(val,       leap) {
    if("" == _MonName[1]) {
        _TimeZone = 9 * 3600
        split("Sun Mon Tue Wed Thu Fri Sat", _WeekName, " ")
        split("Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec",
```

```
                    _MonName, " ")
        split("31 28 31 30 31 30 31 31 30 31 30 31 31 29 31
                    30 31 30 31 31 30 31 30 31", _MonLen, " ")
    }

    if(0 == val + 0) {
        srand()
        Time = srand() + _TimeZone
    } else {
        Time = val
    }

    Sec  = Time % 60
    Min  = int(Time / 60  ) % 60
    Hour = int(Time / 3600) % 24
    Day  = int(Time / (3600 * 24))
    Week = (Day + 4) % 7 + 1
    Year = 1970;
    for(++Day;;) {
        leap =
          ((Year%4)==0 && (Year%100)!=0 || (Year%400)==0) ? 12: 0
        yd   = (leap) ? 366: 365
        if(Day < yd)
            break
        ++Year
        Day -= yd
    }
    for(Mon = 1; _MonLen[Mon+leap]+0 <= Day+0; ++Mon) {
        Day -= _MonLen[Mon+leap];
    }
    return sprintf("%.3s %.3s%3d %02d:%02d:%02d JST %d",
        _WeekName[Week], _MonName[Mon], Day,
        Hour, Min, Sec, Year)
}
```

日本語でどうぞ…

ジャパン・ダイレクトだjpn2awk!!

機　能

「awk もそうだけど、なんでプログラムってほとんど英語でしか書けないんだろう？日本語で書ければもっといいのに」という向きのために作ったのがこのスクリプトだ。本スクリプトを使えば、以下のようなテキストファイルから awk のスクリプトを生成することができるのだ。

リスト 4–4　<sample1.jpn>

```
区切りは「:」で、印字区切りは「,」で、すべて印字してチョ。
```

このファイルの意味は、『区切り文字が ":" のテキストを "," に変更する。』になる。

リスト 4–5　<sample2.jpn>

```
「:」を区切りとする。
「東京都」を見つけたら、2番目「は江戸っ子」と印字する。
「大阪府」を見つけたら、2番目「は浪速の商人」と印字する。
```

このファイルの場合は、『区切り文字が":" のテキストを読み "東京都" を "は江戸っ子" に "大阪府" を "は浪速の商人" に置換する。』の意味になる。

リスト 4–6　<sample3.jpn>

```
「バイトのファイルがあります」を見つけて、しかも、その3番目が「100
0000」以上だったら、3番目に続けて「も使いやがって」と印字して脅せ！
そんでもってよう「バイトが使用可能です.」を見つけて、しかもその1番
目が「5000000」より小さかったら、「もう」1番目「バイトしかねぇじゃ
ねぇか！」って印字するんだ。わかったか!
```

こんなふざけたファイルも、『dir コマンドの出力を調べて文句に変換する。』に解釈される。

操　作

データとして、先に示したような日本語のテキストファイルを指定すると、標準出力に変換結果の awk スクリプトが表示される。

```
A>type sample2.jpn
「:」を区切りとする。
「東京都」を見つけたら、2番目「は江戸っ子」と印字する。
「大阪府」を見つけたら、2番目「は浪速の商人」と印字する。
```

図 4–20　<sample2.jpn>

では、この sample2.jpn を awk のスクリプトに生成させてみよう。

```
A>jgawk -f jpn2awk.awk sample2.jpn
BEGIN {
        FS = ":"
}
/東京都/ {
        print $2, "は江戸っ子"
}
/大阪府/ {
        print $2, "は浪速の商人"
}

A>
```

図 4–21 awk のスクリプトに生成

これをリダイレクションでファイルに落して使用するのも構わないが、以下のように awk を連結しパイプを使ってダイレクトに、できた awk スクリプトを渡してやる方法も面白い。この場合データを標準入力から入れられなくなる点が欠点だ。

ここでは awk 住所録で使用したデータファイルを使用してみた。

```
A>jgawk -f jpn2awk.awk sample3.jpn | jgawk -f - addr.db
藤　英二 は江戸っ子
志村　拓 は江戸っ子
可部　正三 は江戸っ子
岡部　和成 は浪速の商人
          :
          :
```

図 4–22 パイプでスクリプトを渡す

解　説

勘違いしないように最初に言っておくと、なにも本スクリプトが適当に書いた日本語をすべてawkのスクリプトに変換できるわけではない。ある特定の語句（「トークン」と呼ぶ）を認識し、それを適当に解釈して（ここらへんが、なんとも怪しいが）awkスクリプトに変換しているのだ。以下に本スクリプトが認識しているトークンを示す。

トークン	解釈
<数値>	フィールド番号
「<文字列>」	文字列定数
区切り	フィールドセパレータ
印字区切り	出力フィールドセパレータ
(を|が)(見|み) つ	パターン指定
印字	PRINT 命令
が (等しい|同じ)	等号
が (含まれ|あった)	正規表現一致
または|もしくは	論理和
かつ|しかも	論理積
以上	不等号 (以上)
より (大|おお) きい	不等号 (より大きい)
以下	不等号 (以下)
より (小|ちい) さい	不等号 (より小さい)

表 4–2　トークンとその解釈

“印字”トークンは、その前にフィールド番号や文字列定数が書かれていたときはそれらをPRINT命令のパラメータとし、1つもなければ行全体を表示する。また“区切り”と“印字区切り”は、その前に文字列定数があればそれを、なければそれ以降にある文字列定数をフィールドセパレータとして指定する。先に書いた「適当に解釈して」というのは、このあたりの仕様のことだ。

上記のトークン以外の文字は何を書いても無視される。適当に日本語に

なるように文章を書けばよいのだ。

問題点

もっとトークンを増やして、アクションのなかの制御構造なんかも日本語で書けるようにすると実用的になるかもしれない。この場合基本的に日本語は逆ポーランドっぽいので、きちんとトークンスタックを作ってやればなんとかなるかも知れない。ちょうど日本語 mind のような感じになりそうだ。

スクリプト

jpn2awk.awk

```
# jpn2awk
# (日本語からawkスクリプトへの変換)
#
# Copyright (c) 1993 by Taku-SHIMURA, Ken-Washikita
#

BEGIN {
    TokenTbl[TokenTbl+0] = ""                    #  0.トークンなし
    TokenTbl[++TokenTbl] = "「[^」]*」"          #  1.文字列定数指定
    TokenTbl[++TokenTbl] = "[1-9][0-9]*"         #  2.フィールド指定
    TokenTbl[++TokenTbl] = "区切り"              #  3.FS指定
    TokenTbl[++TokenTbl] = "印字区切り"          #  4.OFS指定
    TokenTbl[++TokenTbl] = "(を|が)(見|み)つ"    #  5.パターン指定
    TokenTbl[++TokenTbl] = "印字"                #  6.print指定
    TokenTbl[++TokenTbl] = "が(等しい|同じ)"     #  7.等号
    TokenTbl[++TokenTbl] = "が(含まれ|あった)"   #  8.正規表現一致
    TokenTbl[++TokenTbl] = "または|もしくは"     #  9.論理和
    TokenTbl[++TokenTbl] = "かつ|しかも"         # 10.論理積
    TokenTbl[++TokenTbl] = "以上"                # 11.以上
    TokenTbl[++TokenTbl] = "より(大|おお)き"     # 12.より大きい
    TokenTbl[++TokenTbl] = "以下"                # 13.以下
    TokenTbl[++TokenTbl] = "より(小|ちい)さ"     # 14.より小さい

    FsStrTbl[3]          = "FS"
    FsStrTbl[4]          = "OFS"
```

```
}

{
    line   = $0
    TokenQ = 0
    while ("" != (line = GetToken(line))) {
        if(1 == TokenType || 2 == TokenType) {
            TokenQ[++TokenQ, "type"] = TokenType
            TokenQ[  TokenQ, "val" ] = TokenVal
        } else if(3 == TokenType || 4 == TokenType) {
            fsstr = FsStrTbl[TokenType]
            if(1 == TokenQ[TokenQ, "type"]) {
                fldsep = TokenQ[TokenQ, "val"]
                TokenQ = 0
            } else {
                line   = GetDesigToken(line, 1)
                fldsep = TokenVal
            }
            printf "BEGIN {¥n"
            printf "¥t%s = ¥"%s¥"¥n", fsstr, CutKagi(fldsep)
            printf "}¥n"
        } else if(5 == TokenType) {
            if(0 == PrintValList(0, " || ", "/")) {
                ErrPrint(sprintf("検索文字が見つかりません。(%d)",NR))
                next
            }
        } else if(6 == TokenType) {
            printf " {¥n¥tprint "
            PrintValList(0, ", ", "¥"")
            printf "¥n}¥n"
            TokenQ = 0
        } else if(7 == TokenType) {
            if(0 == PrintValList(2, " == ", "¥"")) {
                ErrPrint("何と何が同じなのか解りません。")
                next
            }
        } else if(8 == TokenType) {
            if(0 == PrintValList(2, " ~ ", "/")) {
                ErrPrint("何に何が含まれているのか解りません。")
                next
            }
        } else if(9 == TokenType) {
            printf " || "
        } else if(10 == TokenType) {
            printf " && "
        } else if(11 == TokenType) {
            if(0 == PrintValList(2, " >= ", "¥"")) {
```

```
                ErrPrint("何より何が大きいのか解りません。")
                next
            }
        } else if(12 == TokenType) {
            if(0 == PrintValList(2, " > ", "¥"")) {
                ErrPrint("何より何が大きいのか解りません。")
                next
            }
        } else if(13 == TokenType) {
            if(0 == PrintValList(2, " <= ", "¥"")) {
                ErrPrint("何より何が小さいのか解りません。")
                next
            }
        } else if(14 == TokenType) {
            if(0 == PrintValList(2, " < ", "¥"")) {
                ErrPrint("何より何が小さいのか解りません。")
                next
            }
        }
    }
}

func GetToken(str,      start, len)
{
    TokenType = 0
    start     = length(str)
    len       = 0
    for(no = 1; no <= TokenTbl; ++no) {
        if (match(str, TokenTbl[no]) && RSTART < start) {
            TokenType = no
            start     = RSTART
            len       = RLENGTH
        }
    }
    TokenVal = substr(str, start, len)
    if(len) str = substr(str, start + len)
    else    str = ""
    return str
}

func GetDesigToken(str, tokenno)
{
    if (match(str, TokenTbl[tokenno])) {
        TokenType = tokenno
        TokenVal  = substr(str, RSTART, RLENGTH)
        str       = substr(str, RSTART + RLENGTH)
    } else {
```

```
        TokenType = 0
        TokenVal  = ""
        str       = ""
    }
    return str
}

func CutKagi(str)
{
    return jsubstr(str, 2, jlength(str) - 2)
}

func PrintToken(no, strsep,             work)
{
    if(no == 0) no = TokenQ
    if(no == 0) return 0
    if(       TokenQ[no, "type"] == 1) {
        work = CutKagi(TokenQ[no, "val"])
        if(work ~ /^[+-]*[1-9][0-9]*$/) strsep = ""
        printf "%s%s%s", strsep, CutKagi(TokenQ[no, "val"]), strsep
    } else if(TokenQ[no, "type"] == 2) {
        printf "$%s", TokenQ[no, "val"]
    }
    return 1
}

func PrintValList(no, sep, strsep,      start, fldno)
{
    if(no == 0) no = TokenQ
    start  = TokenQ - no + 1;
    if(start < 1) {
        TokenQ = 0
        return 0
    }
    for(fldno = start; fldno <= TokenQ; ++fldno) {
        if(start < fldno) printf "%s", sep
        PrintToken(fldno, strsep)
    }
    TokenQ = 0
    return 1
}

func ErrPrint(str)
{
    print str >"/dev/stderr"
    exit 1
}
```

Random Character Stereogram!

立体視 awkだ!!

機　能

ひと昔前に流行った、あの「見える人だけに見えるわけのわからない絵」を書く awk スクリプトだ。ただこのスクリプトで表示されるのは、ランダムな点ではなくランダムな文字である点がいかにも awk らしいところ。立体視を実現する基本的なロジックは、点を使用したものとまったく変わらないので「点」のヤツが見える人には、このスクリプトで作成した変な文字列も、ちゃーんと飛び出して見えるハズ。

RCS の原理

まず RCS(Random Character Stereogram) の原理を説明しよう。図4–23を見て欲しい。

左右の目で、ある物体上の点 (x, y) にある文字 "▽" を見ているとする。このとき、左目と右目で "▽" に対する方向 (視線の角度) は違っている。この方向の違いを脳は遠近感として認識するわけだ。次に目と物体との間にスクリーンを想定し、左右の目と (x, y) をつなぐ視線とスクリーンとが交差する点 x_L, x_Rに同じ文字が表示されているとしよう。

このとき左目で x_L、右目で x_Rに同じ文字を見れば、左右の目は二つの視線の交点、(x, y) にその文字があると勘違いしてしまう。ということは $y = f(x)$ 上の文字と、x_L, x_R上の文字が同じになるようにディスプレイ上に文字を配列すればいいのだ。

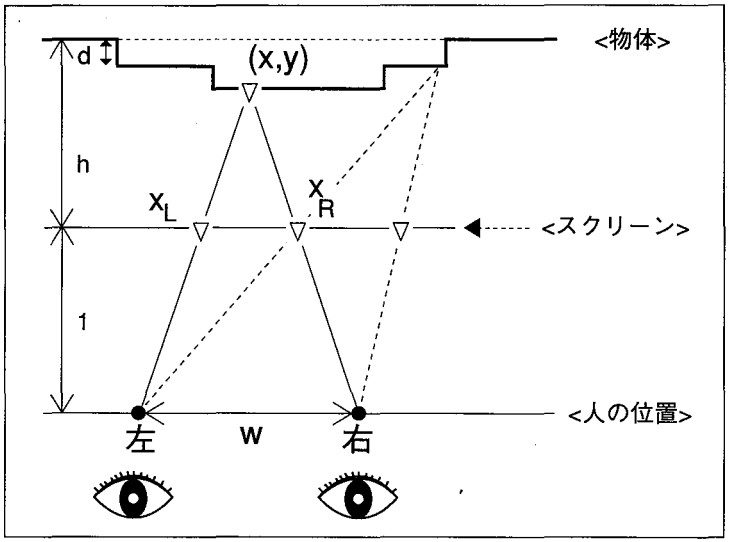


図 4-23 RCS の仕組み

実際にはスクリーン上へ配置する文字を、左側から計算している。まず x_R をもとに、右目からの視線との交点 (x, y) を求め (実はここで手抜きをしている)、それに対応する x_L にある文字を x_R にも置く (左側から置いて行くので、x_L の文字はスクリーン上にあればすでに決まってるはずだ)。スクリーンの両端では、左右どちらかの視線がスクリーン外に出てしまう。左側では対応する x_L がスクリーン上になくなるのでてたらめな文字を置く。

x_R と奥行き y から対応する x, x_L を求める式は次のようになる。

リスト 4-7　文字の配置を求める式

$$x = x_R(1+y) - wy/2$$

$$x_L = x_R wy/(1+y)$$

実を言うと、x を求めるときに y の値を使っているのだ。正確にやるなら、物体を表す方程式 $y = f(x)$ と、右目と x_R を通過する直線の方程式を連立させて解いて x, y を同時に求めるべきなのだが... このスクリプトでは

そこのところを不精して、直前に使った y の値 (つまりちょっと左側の値) を使って、面倒くさい計算をはぶいている。

使用法

まず、立体視に加工するためのデータを用意する。これは図 4–24に示すようなテキストファイルで各点（桁）の飛び出し具合を数値（もちろん半角）で指定するものだ。スクリプトを読んでもらえばわかるが、読み取った 1 桁のデータをそのまま演算に使用しているので数値以外の文字（空白も）の場合は 0 を指定したことと同じになる。ちなみに指定できる最大桁数は 40 桁までだ。次に示すのは今回使用したサンプルデータの 1 例。ここでは 5 角形と星型を組み合わせてみた。

```
00000000000000000022000000000000000000
00000000000000001112211100000000000000
00000000000001111122221111000000000000
00000000011111111222211111111000000000
00000011111111111222222111111111110000
00011111111111112222222211111111111110
01111111111111122222222221111111111111
22222222222222222222222222222222222222
11222222222222222222222222222222222211
01111222222222222222222222222222222110
00111111222222222222222222222222211100
00111111112222222222222222222221111100
00011111111122222222222222221111111000
00011111111222222222222222221111111000
00001111112222222222222222222111111000
00000111122222222222222222222111100000
00000111222222221111112222222211100000
00000011222222211111111112222222211000000
00000001222211111111111111111222210000000
00000002211111111111111111111220000000
00000000000000000000000000000000000000
00000000000000000000000000000000000000
```

図 4–24 RCS サンプルデータ

実　行

データの用意ができたら、以下に示すように本スクリプトにデータを喰わせてやれば、標準出力にわけのわからない文字列が表示される。あとは口をポカーンと開けて、ボーっと眺める（この時、キーボードによだれを垂らさないように気をつけること）。

視線を変えて距離感をずらしていると、多くの人には画面中央に図形が浮き上がって見えるはずだ。

```
A>jgawk -f 3d.awk sample.txt
)n5d%7=b*0)n5d%7=b*0)n5d%7=b*0)n5d%7=b*0)n57%7=b*0)n57%7=b*0)n57%7=b*0)n57%7=b*
13w1sj^$$,13w1sj^$$,13w1sj^$$,13w1sj^$$,3w1jj^$$$,3w1jj^$$$,3w1jj^$$$,3w1jj^$$$
2s04'bj6#=2s04'bj6#=2s04'bj6#=2s04'bj#=2s0'bj##=2s0''bj##=2s0''bj##=2s0''bj##=2
v``yv#b[[/v``yv#b[[/v``yv#b[[/v``y#b[[/v``#b[[[/v``#b[[[[/v``#b[[[[/v``#b[[[[/v
=9s¥u7e((a=9s¥u7e((a=9s¥u7e((a=s¥u7e((a=su7e((aa=su7e((aa==su7e((aa==su7e((aa==
$mlq2rh5=o$mlq2rh5=o$mlq2rh5o$mlq2rh5o$lq2rh5o$llq2rh5o$llq2rhh5o$llq2rhh5o$llq
g&%1&qtd!0g&%1&qtd!0g&%1&td!0g&%1&td!0&%1&td!0&%11&td!0&%11&td!0&&%11&td!0&&%11
2o^m*j!.i12o^m*j!.i12o^m!.i12o^m!.i12o^m!.i12o^m!.i12o^m!.i12o^m!.i12o^m!.i]i
5ol^c3e/bl5ol^c3e/bl5ol^3/bl5ol^3/bl5ol^3/bl5ol^3/bl5ol^3/bl5ol^3/bl5ol^3///b
(yo963,sab(yo963,sab(yo96,sb(yo96,sb(yo96,sb(yo96,sb(yo96,sb(yo96,sb(yyo996,sb(yyo996,s
;(*6l4w*ty;(*6l4w*ty;(*6l4*ty;(6l4*ty;(6l4*ty;(6l4*ty;(6l44*ty;((6l44*ty;((6l44
l//¶lev]ksl//¶lev]ksl//¶]elksl//¶elksl//¶elksl//¶elksl///¶elksl////¶elksl////¶e
]^/f=4c(dx]^/f=4c(dx]^/f=4cdx]^/f=4dx]^/f=4dx]^/f=4dxx]^/f=4dxxx]^/f=4dxxx]^/f=
0h8gueren;0h8gueren;0h8guern;0h8gurn;0h8gurn;0h8gurn;00h8gurn;000h8gurn;000h8gu
]b;z^`!%mrlb;z^`!%mrlb;z^`!%rlb;z`!%rlb;z`!%rlb;z`!%rlb;;z`!%rrlb;;z`!%rrlb;;z`
qs3q^/8l^¶qs3q^/8l^¶qs3q^/8l^qs3^/8l^qs3^/8l^qs3^/8l^qs3^^/8ll^qs3^^/8ll^qs3^^/
aac^0dp=maac^0dp=maac^0dp=aac^0dp=aaa^^0dp=aa^^0dp=aa^^0dp=aa^^^0ddp=aa^^^0ddp=aa^^^
.s*,,`716,.s*,,`716,.s*,,`716,s,,`7166,s,,`7166,s,`7166,s,,,`7166,s,,,`7166,s,,
=atn,`c[xz=atn,`c[xz=atn,`c[xz=t,,cc[xz=t,,cc[xz=t,,c[xz=ttt,,c[xz=ttt,,c[xz=tt
mp%kr'bv0)mp%kr'bv0)mp%kr'bv0)mkkr'bv0)mkkr'bv0)mkkr'bv0mkkkkr'bv0mkkkkr'bv0mkk
[ga];-o/'6[ga];-o/'6[ga];-o/'6[ga];-o/'6[ga];-o/'6[ga];-o/'6[ga];-o/'6[ga];-o/'
p2=gz/paa9p2=gz/paa9p2=gz/paa9p2=gz/paa9p2=gz/paa9p2=gz/paa9p2=gz/paa9p2=gz/paa

A>
```

図 4–25　RCS の実行例

注 意

このスクリプトでは 14 インチのディスプレイのサイズを前提として計算しているので、紙に打ち出しても見ることはできるが、基本的にディスプレイ上で見てもらいたい。

どうしても立体的に見えない人は、視力にも原因があるかもしれない。左右の視力に差があると見えにくいようだ。同様の理由で、利き目の方でモノを見てしまう癖の強い人も見えにくい（そんなこと言われても…）。要するに左右の目で均等に見ることがコツのようだ。

また、データによっても見えにくいものがある。あまり細かいデータを作ってしまうとよくわからない。データを作る時のコツとしては、ある程度の面積のあるデータを作成することだ。

スクリプト

RCS.AWK

```
BEGIN {
  sw = 40;       # 元のパターンの幅
  dw = 78;       # 生成される RCS の幅
  hdw = dw / 2;
  w = 20;        # 両目の幅
  h = 1;         # 画面と基準面の距離（目と画面の距離が単位）
  d = .2;        # 単位当たりの浮き上がり方（//）
  ss = "abcdefghijklmnopqrstuvwxyz0123456789!#$%^&*()-=¥¥[];'`,./";
  srand();
}
{
  xr = -hdw; y = h * 1.0; maxxl = -999;
  s = "";
  while (xr < hdw) {
    x = xr * (1 + y) - y * w / 2;
    i = x / (1 + h) + sw / 2 + 1;
    c = i < 1 ? 0 : substr($0, i, 1);
    y = h - d * c;
    xl = xr - w * y / (1 + y);
    if (xl < -hdw || xl >= hdw || xl <= maxxl) {
        c = substr(ss, rand() * length(ss) + 1, 1);
```

```
    } else {
        c = substr(s, xl + hdw + 1, 1);
        maxxl = xl;
    }
    s = s c;
    xr++;
  }
  print s;
}
```

ビジネス活用度 NO.1!!

awk 表プロセッサ

機　能

タブまたはスペースで区切って書かれたテキストを表に仕立ててくれるスクリプト。単純なだけに最も活用範囲の広いスクリプトとなるだろう。

操　作

図 4-26のように、表にしたい項目をタブまたは空白で区切って書いたデータを用意する。ちなみにこの時に空行を入れておくとその行は太線になって表示される。

```
本体          AUVA/460     245,000
CRT           MF8217       115,000
音源ボード    SB16ASP       35,000
5"FDD    不明      9,800

合計     --            404,800
```

図 4-26 <dosv.txt>

40まんえん!! うーむ…

各フィールドは自動判別され、数字とカンマからなるフィールドは右詰め、それ以外のフィールドはすべて左詰めにされる。実際のビジネスの世界で使うためにはまだ機能不足だが（UNIX の tbl などでは金額専用のフィールド指定があるほどだ）。

また項目名など、フィールド内でセンタリング（中央揃え）したいケースもあるだろう。その場合、丸括弧

“( )”で囲めばセンタリング

される。丸括弧で囲んだもの自体をセンタリングしたい場合は…? ただ二重に囲めばよい。

データファイルができたら、それを本スクリプトに喰わせると綺麗な表に変換されて標準出力に出力されるわけだ。

```
A>jgawk -f hyou.awk dosv.txt
+==========+========+=======+
¦本体      ¦ALUA/460¦245,000¦
+----------+--------+-------+
¦CRT       ¦MF8217  ¦115,000¦
+----------+--------+-------+
¦音源ボード¦SB16ASP ¦ 35,000¦
+----------+--------+-------+
¦5"FDD     ¦不明    ¦  9,800¦
+----------+--------+-------+
¦          ¦        ¦       ¦
+----------+--------+-------+
¦合計      ¦--      ¦404,800¦
+==========+========+=======+
```

図 4–27 <hyou.awk>の実行

表の罫線はオプションによって、“–a” で ASCII コード、“–j” で JIS 罫線、“–n” で NEC 固有罫線という具合に 3 つの文字コードの中から選択することができるようになっている。

```
A>jgawk -f hyou.awk -- -n dosv.txt
```

本体	AUVA/460	245,000
CRT	MF8217	115,000
音源ボード	SB16ASP	35,000
5"FDD	不明	9,800
合計	--	404,800

```
A>
```

図4-28 NEC罫線を使用

解 説

前の図4-28を見て「あれ？」と思った向きは鋭い。本スクリプトではオプションのパラメータを"-?"という形で指定するようになっているが、単にデータファイル名の前に"-?"という風に指定すると、

awkが自分のオプションとして認識

してしまいスクリプトにパラメータとして渡らない（つまりARGVシステム変数に入らない）。そこで"--"というawkのオプションの出番となる。このオプションは、「それ以降"-"で始まるパラメータがあっても、それはawkのオプションではなく、ファイル名として認識するヨ」というものなのだ。この

"--"オプションを先に指定

することによって、"-n"をスクリプトにパラメータとして引き渡すことを

実現しているわけだ。

まあ、スクリプトのオプションスイッチとして使用する文字を“=”等で始めるようにすれば特に問題はないのであるが…。あえて“–”をスイッチ文字にしたのは、「バッチやシェルスクリプトで本スクリプトを実行させる場合を考えると、スイッチ文字の標準である“–”を使いたい」、ということからだ。

オプション

表の罫線はオプションによって“–a”でASCIIコード、“–j”でJIS罫線、“–n”でNEC固有罫線という具合に3つの文字コードの中から選択することができるようになっている（オプションを省略した場合ASCIIコードとなる）。

オプション	出力される文字コード	対応機種
–a（デフォルト）	ASCIIコード	DOS/V,PC98,EPSONなどほとんどすべて
–j	JIS罫線	DOS/V,EPSONなど多数
–n	NEC固有罫線	PC98,EPSON

表4–3　オプションおよび対応機種

問題点

この仕様では、括弧で囲まれた文字を左寄せで表示したいとか、文字の両側にある程度の空白を入れたいなどの細かい指定ができていない。

また、

複数の行や欄をぶち抜きで表示

できたりすると便利だろうなあ…（LaTeX だとここらへんの指定が面倒なので）とも思うが、まあ元のデータに手を掛けずに表を作る場合このあたりが限界かもしれない。

この他に文字コードの問題がある。PC9800 シリーズは JIS で定められた罫線符号に対応していない。このため "-j" オプションをつけた場合は罫線を画面上で確認できない。逆に PC9800 シリーズ独自の罫線文字を持っている。これに対応した "-n" オプションもあるのだが、今度は PC9800 シリーズとその互換機以外では見ることができなくなってしまう。

ちなみに区切り文字の変更などは以下のようにオプションをつけて設定してやれば可能である。

リスト 4-8　区切り文字を ":" に設定する

```
jgawk -F : -f hyou.awk  foo
```

スクリプト

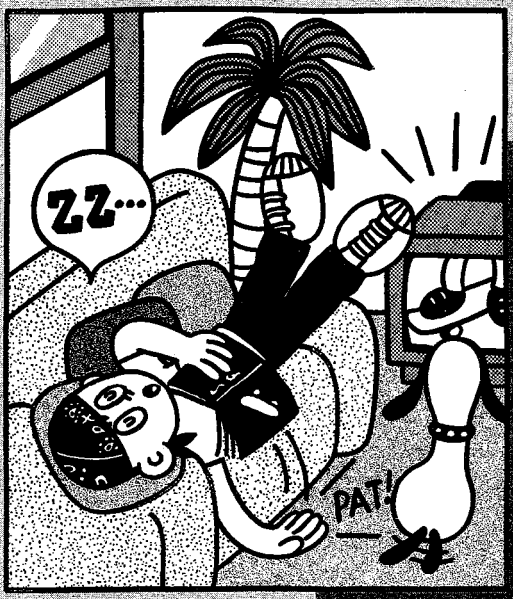
HYOU.AWK

```
BEGIN {
    nf = 0;
    split("+ + + + + +", right_char);
    split("+ + + + + +", left_char);
    split("+ + + + + +", x_char);
    split("- = - = - =", hline_char);
    split("| |", vline_char);
    width = 1;
    for (i = 1; i < ARGC && ARGV[i] ~ /^-/; i++) {
        if (ARGV[i] == "-a") { #dummy
        } else if (ARGV[i] == "-n") {

#           split("┌ ┌ ├ ├ └ └", left_char);
            split("┏ ┏ ┣ ┣ ┗ ┗", left_char);
#           split("┐ ┐ ┤ ┤ ┘ ┘", right_char);
            split("┓ ┓ ┫ ┫ ┛ ┛", right_char);
            split("┬ ┬ ┼ ┼ ┴ ┴", x_char);
            split("─ ━ ─ ━ ─ ━", hline_char);
            split("│ ┃", vline_char);
            width = 2;
        } else if (ARGV[i] == "-j") {
            split("┏ ┏ ┣ ┣ ┗ ┗", left_char);
            split("┓ ┓ ┫ ┫ ┛ ┛", right_char);
            split("┬ ┬ ┼ ┼ ┴ ┴", x_char);
            split("─ ━ ─ ━ ─ ━", hline_char);
            split("│ ┃", vline_char);
            width = 2;
        } else {
            usage();
            exit(-1);
        }
        delete ARGV[i] ;
    }
}

{
    buffer[n++] = $0;
    for (i = 1; i <= NF; i++) {
        len = length($i);
```

```
        if ($i ~ /^¥(.*¥)$/) {
            len -= 2;
        }
        if (col[i] < len) {
            col[i] = len;
        }
    }
    nf = (nf < NF) ? NF : nf;
}

END {
    if (escape) exit escape;

    for (i = 1; i <= nf; i++) {
        col[i] += col[i] % width;
    }
    for (i = 1; i <= nf; i++) {
        left_fmt[i] = sprintf("%%-%ds", col[i]);
        right_fmt[i] = sprintf("%%%ds", col[i]);
    }

    for (i = 0; i < n && buffer[i] == ""; i++);
    if (i == n) exit;

    hs = 2;
    while (i < n) {
        hline(hs);

        split(buffer[i], field);
        for (j = 1; j <= nf; j++) {
            vline((j == 1) + 1);
            if (field[j] ~ /^¥(.*¥)$/) {
                field[j] = substr(field[j], 2, length(field[j]) - 2);
                printf "%s", center(col[j], field[j]);
            } else if (field[j] ~ /^[¥¥$￥]?-?[0-9,.]+円?$/) {
                printf right_fmt[j], field[j];
            } else {
                printf left_fmt[j], field[j];
            }
        }
        vline(2);
        printf "¥n";

        hs = 3; i++;
        while (i < n && buffer[i] == "") {
            hs = 4; i++;
        }
```

```
    }
    hline(6);
}

function hline(kind     ,i, j) {
    for (i = 1; i <= nf; i++) {
        printf "%s", (i == 1) ? left_char[kind] : x_char[kind];
        for (j = 1; j <= col[i] / width; j++) {
            printf "%s", hline_char[kind];
        }
    }
    printf "%s¥n", right_char[kind];
}

function vline(kind) {
    printf "%s", vline_char[kind];
}

function center(width, item,        ilen, llen)
{
    ilen = length(item);
    if (ilen > width) { # unreachable
        return substr(item, width);
    }
    llen = int((width - ilen) / 2);
    return repeat(" ", llen) item repeat(" ", width - ilen - llen);
}

function repeat(s, n,       r)
{
    r = "";
    while (n-- > 0) {
        r = r s;
    }
    return r;
}

function usage() {
    printf "usage: %s -f hyo.awk [-- -ajn] <filename>¥n",
     ARGV[0] > "/dev/stderr";
    escape = 1; exit;
}
```

APPENDIX



APPENDIXの構成

このAPPENDIXは、

- 文法のまとめ　(7項目)
- 関数リファレンス　(abc順)
- 組み込み変数リファレンス　(abc順)

から構成されている。

バージョンについて

このAPPENDIXは“gawk 2.15.2”および“jgawk 2.15.2+1.0(SJIS)”をもとに作成した。したがってそれ以外のバージョンのawkを使う場合、動作の違いが生じることがある。

表記について

リファレンスでの表記方法は以下のとおり

表記方法	意味
斜体	数値変数 (例:x)
<...>	定数または変数
[...]	括弧内は省略可能
{a,b}	aまたはb

文法のまとめ

文字列定数

定　数	文字コード	意　味
¥a	0x07	警告 (ベル)
¥b	0x08	バックスペース
¥f	0x0c	改頁 (フォームフィード)
¥n	0x0a	改行 (ニューライン)
¥r	0x0d	キャリッジリターン
¥t	0x09	水平タブ
¥v	0x0b	垂直タブ
¥x*hh*	–	16 進数 *hh* で表現される文字
¥*ooo*	–	8 進数 *ooo* で表現される文字
¥*c*	–	上記以外の文字 *c* そのもの

定義済みファイル名

ファイル名	意　味
/dev/null	どこからも入力されない、どこにも出力されない
/dev/tty	コンソール出力
/dev/stdin	標準入力 ("/dev/fd/0" と同じ)
/dev/stdout	標準出力 ("/dev/fd/1" と同じ)
/dev/stderr	標準エラー出力 ("/dev/fd/2" と同じ)
/dev/fd/*n*	ファイル番号 *n* で表されるファイル
/dev/pid	自分の PID が読み出せる (gawk 2.15 以降)
/dev/ppid	親プロセスの PID が読み出せる (gawk 2.15 以降)
/dev/pgrpid	自分のグループ ID が読み出せる (gawk 2.15 以降)

ファイル名	意　味
/dev/user	システムコール getuid、geteuid、getgid、getegid、getgroups の返り値が、それぞれフィールドに区切られて読み出せる。ただし getgroups はサポートされている場合のみである (gawk 2.15 以降)

・注意 1　MS-DOS 版では "/dev/pid"、"/dev/ppid"、"/dev/pgrpid" はそれぞれ "PSP のセグメントアドレス"、"親プロセスの PSP のセグメントアドレス"、"常に 0" を返す。また "/dev/user" は 4 つのフィールドに 0 を格納して返す。

・注意 2　MS-DOS 版の jgawk 2.11.1 + 3.0 以前では "/dev/null" および "/dev/tty" を指定しても OS 上の問題から正常に動作しない。しかし MS-DOS 版の jgawk 2.15.2 + 1.0 ではこれらについても正常に動作するようになっている。

正規表現

正規表現が適合する文字列は以下のとおり

メタ文字	意味
c	文字 c (ただし正規表現で意味をもたない文字)
¥c	文字 c そのもの
.	改行以外の任意の一文字
^	行頭
$	行末
[abc...]	文字 abc... のうちの任意の一文字
[^abc...]	文字 abc... と改行以外の任意の一文字
(r)	正規表現 r (グループ化)
(r1)|(r2)	正規表現 r1 または正規表現 r2 (選択)
(r1)(r2)	正規表現 r1 つづけて正規表現 r2 (連結)
(r)+	1 回以上の正規表現 r
(r)*	0 回以上の正規表現 r
(r)?	0 回または 1 回の正規表現 r

正規表現中の括弧は演算子の優先順位が守られている場合、省略可能。

演算子 （式演算子）

演算子	意味
$	フィールド参照
++ --	インクリメント、デクリメント
^	累乗
+ - !	単項演算子
*/ %	積、商、剰余
+ -	和、差
(空白)	文字列連接 (ex. “ABC” “DEF”)
<<= >>= != ==	関係演算子
~ !~	正規表現適合、非適合
in	配列要素
&&	論理積
||	論理和
?:	三項演算子
= += -= *= /= %= ^=	代入

パターン

パターン	意味
BEGIN	一番最初
END	一番最後
式	式が真となる行のみ
式1, 式2	式1が真となる行から式2が真になる行まで

パターンでは、awk で式として評価されるものはすべて許容される。したがって

```
!/^#/
/^[ ¥t]/ && /[^ ¥t]$/
```

このようなパターンも許される。なおパターンを省略するとすべての行が対象になる。

制御文

制御文	意味
if (式) 文1 [else 文2]	式が真ならば文1を実行、偽ならば文2を実行する
while (式) 文	式が真の間、文を実行する
do 文 while (式)	文を実行し式が真ならば再度文を実行する
for (式1; 式2; 式3) 文	式1が初期値、式3が増分のとき式2が真の間、文を実行する
for (変数 in 配列) 文	変数に連想配列の添字がひとつづつ入り、文を実行する
break	while 文、または for 文を終了する
continue	while 文、または for 文で次の繰り返しを開始する
next	スクリプトの最初に戻り、次のレコードを読み込む
next file	スクリプトの最初に戻り、次のファイルの先頭レコードを読み込む（2.14 以降）
exit [式]	スクリプトを終了する。END ルールがあればそれを実行する
return [式]	ユーザ定義関数の終了。式があれば、その値を返す

入出力文

スクリプト中で何も指定しない場合、getline 文の入力は処理中の入力ファイル(組み込み変数 FILENAME の値)または標準入力('/dev/stdin')から読み込まれ、print 文や printf 文の出力は標準出力('/dev/stdout')に書き込まれる。これらは以下のように指定することによって入力元および出力先を変更することができる。

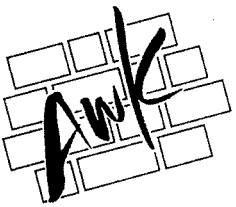
書式	意味
print > 〈ファイル〉	出力は〈ファイル〉に上書きされる
print >> 〈ファイル〉	出力は〈ファイル〉に追加される
print | 〈コマンド〉	出力は〈コマンド〉の標準入力として渡される
getline < 〈ファイル〉	入力は〈ファイル〉から読み込まれる
〈コマンド〉 | getline	入力は〈コマンド〉の標準出力が読み込まれる

〈ファイル〉や〈コマンド〉は変数や式で指定することもできる。たとえば

```
print > "section." number
```

と指定することもできる。

また、awk では同時に読み書きすることのできる〈ファイル〉や〈コマンド〉の数が限られている。必要のなくなった〈ファイル〉や〈コマンド〉は close 文で閉じることができる。ただし close 文の引数には〈ファイル〉または〈コマンド〉を開いたときと同じ文字列を指定しなければならない。



リファレンス

リファレンスの構成

- このリファレンスでは、関数と組み込み変数について取り扱う
- 関数については、「文法のまとめ」の制御文、入出力文の項も参考にしてほしい
- リファレンスでの表記方法は以下のとおり

関数/変数の種別 ——→ 算術関数

関数/変数名 ——→ **atan2**

関数/変数の書式 ——→ 書　式

atan2(*y*, *x*)

機能の解説 ——→ 機　能

y/x の逆正接をラジアン ($-\pi < atan2() <= \pi$) で返す。

実際の使用例 ——→ サンプルプログラム

```
BEGIN {
  PI = atan2(0, --1)
  print "#input Y and X"
}
```

使用上注意すべきポイント——→ 注　意

atan2() の戻り値はコンパイルした処理系に依存する。

参照すべき項目 ——→ 参　照

cos, sin

算術関数

atan2

書　式

atan2(y, x)

機　能

y/x の逆正接をラジアン ($-\pi < atan2() <= \pi$) で返す。

サンプルプログラム

```
BEGIN {
    PI = atan2(0, -1)
    print "# input Y and X"
}

{
    if (NF == 0) {
        exit
    } else if (NF != 2) {
        print "# input Y and X"
    } else {
        printf " %6.1f [DEG]¥n", atan2($1, $2) / PI * 180
    }
}
```

注　意

atan2() の戻り値はコンパイルした処理系に依存する。

参　照

cos, sin

その他

close

書 式

close({ 〈ファイル〉 , 〈コマンド〉 })

機 能

awk によって開かれたファイルまたはパイプ(コマンド)を閉じる。〈ファイル〉 または 〈コマンド〉 には、それが開かれたときと同じ文字列を指定しなければならない。

サンプルプログラム

```
BEGIN {
    printf "avilable: %s bytes.¥n", dfree()
    # 2回やっても大丈夫！
    printf "avilable: %s bytes.¥n", dfree()
}

function dfree(s, a) {
    while ("dir" | getline s) {
        if (s ~ /使用可能です¥./) {
            break
        }
    }
    close("dir")
    split(s, a, " ")
    return a[1]
}
```

参 照

getline, print, printf, 文法のまとめ（入出力文）

算術関数

COS

書 式

cos(*x*)

機 能

*x*の余弦を返す。*x*の値はラジアンで与える。

サンプルプログラム

```
BEGIN {
    WIDTH2  = 10
    HEIGHT2 = 10
    PI      = atan2(0, -1)

    for (y = 0; y <= WIDTH2 * 2; y++) {
        n = int((1 + cos(y / WIDTH2 * PI)) * HEIGHT2 + 0.5)
        s = ""
            for (x = 0; x <= HEIGHT2 * 2; x++) {
            if (x == n) {
                s = s "*"
            } else if (x == HEIGHT2) {
                s = s "|"
            } else {
                s = s " "
            }
        }
        print s
    }
}
```

参 照

atan2, sin

その他

delete

書　式

delete 〈配列の要素〉

機　能

配列の要素を削除する。

サンプルプログラム

```
BEGIN {
    delete ARGV[0]
    if (0 in ARGV) {
        print "Can't delete..." >"/dev/stderr"
        exit
    }
    for (i in ARGV) {
        printf "%2d: %s¥n", i + 0, ARGV[i]
    }
}
```

算術関数

exp

書　式

exp(x)

機　能

定数 e に対する指数関数 (e^x) の値を返す。x の値が扱える範囲を超えた場合、実行時エラーとなる。なお、扱える範囲は処理系により異なる。

サンプルプログラム

```
BEGIN {
    printf "# input number¥n"
}

NF == 0 {
    exit
}

{
    printf "exp(%d) = %g¥n", $1, exp($1)
}
```

参　照

log, sqrt

入出力文

getline

書　式

getline [〈変数〉] [< 〈ファイル〉]

または

〈コマンド〉 | getline [〈変数〉]

機　能

現在処理中の入力ファイル(組み込み変数 FILENAME の値) から読み込み、組み込み変数 NR および FNR の値を変更する。ただし "|","<" を使う事により、他のファイルから入力する事が出来る。この場合 NR および FNR の値は変更されない。

〈変数〉 を省略した場合、代わりに$0 に格納される。この場合、入力行はフィールドに分割され、組み込み変数 NF に分割されたフィールドの数が格納される。また、各フィールドは$1～$NF に格納される。

読み込みに成功した場合は 1 を、〈ファイル〉 の終わりに達した場合は 0 を、読み込みに失敗した場合は−1 をそれぞれ返す。

サンプルプログラム

```
BEGIN {
    print cwd()
}

function cwd(s) {
    "chdir" | getline s
    close("chdir")
    gsub(/¥¥/, "/", s)
    return tolower(s)
}
```

参　照

NF, NR, FNR, close, 文法のまとめ（入出力文）

文字列関数

gsub

書　式

gsub(〈正規表現〉,〈置換文字列〉[,〈対象文字列〉])

機　能

〈対象文字列〉中で〈正規表現〉に一致する部分文字列をすべて〈置換文字列〉に置換し、置換された数を返す。〈対象文字列〉を省略した場合、代わりに$0が使われる。

〈置換文字列〉中の'&'は、〈正規表現〉に一致した部分文字列に置き換えられる。〈置換文字列〉に'&'そのものを含めたい場合、'¥¥&'と書かなければならない。

subは〈正規表現〉に最初に一致した部分文字列のみを置換するのに対し、gsubは一致した部分文字列すべてを置換する。つまりgsubの'g'は'global'の意味である。

サンプルプログラム

```
{
    gsub(/¥<PPP¥>/, "& is Programming Products")
    print
}
```

注　意

gsubおよびsubは他の関数と異なり、引数(〈対象文字列〉)に変更を加える。したがって〈対象文字列〉は変数でなくてはならない。次の例では実行時エラーとなる。

```
gsub(/ASCII/, "AhSKI!", "ASCII Corp.");
```

参　照

sub

文字列関数

index

書 式

index(〈対象文字列〉,〈部分文字列〉)

機 能

〈対象文字列〉中の〈部分文字列〉の位置をバイト単位で返す。〈対象文字列〉が〈部分文字列〉を含まない場合0を返す。漢字コードには対応していないので注意すること。

サンプルプログラム

```
function htod(s, d, i, j) {
    d = 0
    for (i = 1; i <= length(s); i++) {
        if ((j = index("0123456789ABCDEF", substr(s, i, 1))) == 0) {
            return -1
        } else {
            d = d * 16 + j - 1
        }
    }
    return d
}

{
    print htod($0)
}
```

参 照

jindex, length, substr, match

算術関数

int

書　式

int(*x*)

機　能

実数値 *x* の整数部分を返す。その値は 0 から *x* の範囲にある、もっとも *x* に近い整数値となる。たとえば int(1.5) は 1 を返し、int(–1.5) は–1 を返す。

サンプルプログラム

```
BEGIN {
    print int(1.5)
    print int(-1.5)
}
```

注　意

jgawk 2.11.1 + 2.?(MS–DOS, SJIS) では、実数値 *x* を超えない最大の整数値が返される。これは *x* が負の数だったときの結果が異なることを意味する。つまり int(–1.5) の結果が–2 になってしまう。なおこれは jgawk 2.11.1 + 3.0(MS–DOS, SJIS) 以降で修正されている。

文字列関数、漢字対応

jindex

書 式

jindex(〈対象文字列〉,〈部分文字列〉)

機 能

〈対象文字列〉中の〈部分文字列〉の位置を文字単位で返す。漢字コードに対応しており、漢字を1文字として数える。〈対象文字列〉が〈部分文字列〉を含まない場合、0を返す。

サンプルプログラム

```
BEGIN {
    print jindex("いBは", "B")
    print index("いBは", "B")
}
```

参 照

index, jlength, jsubstr

文字列関数、漢字対応

jlength

書　式

jlength([〈対象文字列〉])

機　能

〈対象文字列〉の長さを文字数で返す。漢字コードに対応しており、漢字を1文字として数える。〈対象文字列〉を省略した場合、代わりに$0が使われる。

サンプルプログラム

```
BEGIN {
    print jlength("いBは")
    print length("いBは")
}
```

参　照

length, jindex, jsubstr

文字列関数、漢字対応

jsubstr

書 式

jsubstr(〈対象文字列〉 , 〈開始位置〉 [, 〈文字数〉])

機 能

〈対象文字列〉中の文字単位で指定した〈開始位置〉から〈文字数〉分の部分文字列を返す。漢字コードに対応しており、漢字を１文字として数える。〈文字数〉を省略した場合、〈対象文字列〉の〈開始位置〉から最後までを返す。

サンプルプログラム

```
{
    sub(/^[ ¥t]+/, "")
    $0 = carry $0
    while (length($0) > 72) {
        print (s = jfold($0, 72))
        $0 = jsubstr($0, jlength(s) + 1)
    }
    carry = $0
}
END {
    print carry
}
function jfold(s, n, t) {
    t = substr(s, 1, n + 1)
    if (length(t) > n) {
        t = jsubstr(s, 1, jlength(t) - 1)
        if (jindex("‘<(‘", jsubstr(s, jlength(t), 1)))
            return jsubstr(s, 1, jlength(t) - 1)
        if (jindex("’>)、。—’",jsubstr(s,jlength(t)+1,1)))
            return jsubstr(s, 1, jlength(t) + 1)
    }
    return t
}
```

参 照

substr, jindex, jlength

文字列関数

length

書　式

length([〈対象文字列〉])

機　能

〈対象文字列〉の長さをバイト数で返す。〈対象文字列〉を省略した場合、代わりに$0が使われる。漢字コードには対応していないので注意すること。

サンプルプログラム

```
fname == "" {
    fname = FILENAME
}
fname != FILENAME {
    printf "%7d %7d %7d %s¥n", line, word, char, fname
    fname = FILENAME
    line = word = char = 0
}
{
    line += 1
    word += NF
    char += length()
    tword += NF
    tchar += length()
}
END {
    printf "%7d %7d %7d %s¥n", line, word, char, fname
    if (ARGC > 2) {
        printf "%7d %7d %7d total¥n", NR, tword, tchar
    }
}
```

参　照

jlength, index, substr

算術関数

log

書 式

log(*x*)

機 能

e を底とする *x* の対数関数 ($\log_e x$) の値を返す。*x* の値は 0 より大きくなければならない。もし 0 以下の値を与えると、実行時エラーとなる。

サンプルプログラム

```
BEGIN {
    WIDTH = 75

    for (i = 1; i <= 10; i++) {
        while (x < int(log(i) / log(10) * WIDTH + 0.5)) {
            s1 = s1 "-"
            s2 = s2 " "
            x++
        }
        s1 = s1 "+"
        s2 = s2 i
        x++
    }
    print s1
    print s2
}
```

参 照

exp, sqrt

文字列関数

match

書　式

match(〈対象文字列〉,〈正規表現〉)

機　能

〈対象文字列〉中で〈正規表現〉に一致する位置をバイト単位で返し、一致した部分文字列の位置と長さを組み込み変数 RSTART と RLENGTH に格納する。一致するものがない場合 0 を返す。

サンプルプログラム

```
{
    if (match($0, /0x[0-9A-F]+/) != 0) {
        print substr($0, RSTART, RLENGTH)
    }
}
```

参　照

index

入出力文

print

書式

print([〈式の並び〉])
[{> 〈ファイル〉 ,>> 〈ファイル〉 ,| 〈コマンド〉 }]

機能

〈式の並び〉を出力し、最後にORS（デフォルト値は改行）を出力する。〈式の並び〉はコンマ“,”で区切って複数指定でき、出力時は各項目の間がOFSになる。

また〈式の並び〉を省略した場合、代わりに$0の内容が出力される。出力は標準出力に行われる。ただし“>”, “>>”, “|”を使用することで、出力先を変更することができる。

printの“(”と“)”は、〈式の並び〉で関係演算子などが使われてなければ省略可能である。

サンプルプログラム

```
{
    print NR, $1 >"/dev/stderr"
}
```

参照

OFS, OR, OFMT, printf, close, 文法のまとめ（入出力文）

入出力文

printf

書　式

printf(〈書式〉 [, 〈式の並び〉])
　[{> 〈ファイル〉 ,>> 〈ファイル〉 ,| 〈コマンド〉 }]

機　能

〈式の並び〉 を 〈書式〉 にしたがって変換し出力する。〈式の並び〉 はコンマ “,” で区切って 〈書式〉 に対応した数だけ指定する。出力は標準出力に行われる。ただし “>”, “>>”, “|” を使用することで、出力先を変更することができる。

printf の “(” と “)” は、〈式の並び〉 で関係演算子などが使われていなければ省略可能である。

〈書式〉 の指定形式は以下のとおり。

%[<フラグ>] [<幅>] [.<桁>] <形式>

書式	内容
<フラグ>	出力する位置や符号、小数点などを指定する。許される組み合わせであればひとつの<形式>に対して複数の<フラグ>を指定できる
<幅>	出力する幅を指定する
<桁>	文字列を変換する場合は最大幅を表す。数値を変換する場合は小数点以下の桁数を表す
<形式>	変換の方法を指定する
	<幅>および<桁>に “*” を指定した場合、<式の並び>の中で<幅>および<桁>を指定することができる (gawk 2.13 以降)

<形式>で指定できるものは次のとおりである。

形式	意味
c	文字列の先頭の 1 文字
d	10 進数 (整数部分のみ)
i	d と同じ
e	浮動小数点形式 ([−]d.dddddd E[+−]dd)
f	浮動小数点形式 ([−]ddd.dddddd)
g	%e、%f のうち短い方で、末尾の 0 は出力されない
o	符号なし 8 進数
s	文字列
x	符号なし 16 進数 (0〜f)
X	符号なし 16 進数 (0〜F)
%	"%" そのもの (<式の並び>は必要としない)

<フラグ>で指定できるものは次のとおりである。

フラグ	内容
0	右詰めで出力する。<幅>に満たない場合、先頭に "0" が足される。文字列を変換する場合は無視される
−	左詰めで出力する。<幅>に満たない場合、末尾に空白 (" ") が足される。
+	変換結果が符号つきの場合、直前に "+" または "−" が出力される。(gawk 2.13 以降)
空白	変換結果が符号つきの場合、直前に空白 (" ") または "−" が出力される。(gawk 2.13 以降)
#	<形式>の o、x、X とともに使うと直前に "0"、"0x"、"0X" が出力される。(gawk 2.13 以降)
	"0" と "−" ともに指定しない場合、右詰めで出力される。<幅>に満たない場合、先頭に空白 (" ") が足される。

サンプルプログラム

```
BEGIN {
    e  = exp(1)
    pi = atan2(0, -1)
    printf "%%f¥t:¥"%f¥"¥t¥t%%s¥t:¥"%s¥"¥n",pi,e
    printf "%%10f¥t:¥"%10f¥"¥t¥t%%10s¥t:¥"%10s¥"¥n",pi,e
    printf "%%-10f¥t:¥"%-10f¥"¥t¥t%%-10s¥t:¥"%-10s¥"¥n",pi,e
    printf "%%10.5f¥t:¥"%10.5f¥"¥t¥t%%10.5s¥t:¥"%10.5s¥"¥n",pi,e
    printf "%%-10.5f¥t:¥"%-10.5f¥"¥t¥t%%-10.5s¥t:¥"%-10.5s¥"¥n",pi,e
    printf "%%010.5f¥t:¥"%010.5f¥"¥t¥t%%010.5s¥t:¥"%010.5s¥"¥n",pi,e
}
```

参　照

sprintf, print, close, 文法のまとめ（入出力文）

算術関数

rand

書 式

rand()

機 能

0以上1未満の範囲で、疑似乱数を返す。

サンプルプログラム

```
BEGIN {
    srand()
    for (i = 1; i <= 4; i++) {
        n[i] = int(rand() * 10)
        for (j = 1; j < i; j++)
            if (n[i] == n[j])
                i--
    }
}

{
    h = b = 0
    for (i = 1; i <= 4; i++) {
        if ($i == n[i])
            h++
        else
            for (j = 0; j < 3; j++)
                if ($i == n[(i + j) % 4 + 1])
                    b++
    }
    printf "# H:%d B:%d¥n", h, b
}

END {
    print n[1], n[2], n[3], n[4]
}
```

参 照

srand

算術関数

sin

書　式

sin(*x*)

機　能

*x*の正弦を返す。*x*の値はラジアンで与える。

サンプルプログラム

```
BEGIN {
    WIDTH2  = 39
    HEIGHT2 = 10
    PI      = atan2(0, -1)

    for (x = 0; x <= WIDTH2 * 2; x++) {
        n[x] = int((1 - sin(x / WIDTH2 * PI)) * HEIGHT2 + 0.5)
    }
    for (y = 0; y <= HEIGHT2 * 2; y++) {
        s = ""
        for (x = 0; x < WIDTH2 * 2 + 1; x++) {
            if (n[x] == y) {
                s = s "*"
            } else if (y == HEIGHT2) {
                s = s "-"
            } else {
                s = s " "
            }
        }
        print s
    }
}
```

参　照

atan2, cos

文字列関数

split

書 式

split(〈対象文字列〉,〈配列〉[,〈正規表現〉])

機 能

〈対象文字列〉を〈正規表現〉で分割し〈配列〉に格納する。戻り値として分割数を返す。ただし、対象文字列に空文字を指定した場合は0となる。〈正規表現〉を省略した場合、代わりに組み込み変数FSが使われる。

サンプルプログラム

```
BEGIN {
    split("一,二,三,四,五,六,七,八,九,十", num, ",")
    for (i = 1; i <= 10; i++) {
        printf "%2d : %s¥n", i, num[i]
    }
}
```

参 照

FS

文字列関数

sprintf

書　式

sprintf(〈書式〉 [,〈式の並び〉])

機　能

〈式の並び〉 を printf と同様の〈書式〉にしたがって変換した文字列を返す。〈式の並び〉はコンマ “,” で区切って〈書式〉に対応した数だけ指定する。

サンプルプログラム

```
BEGIN {
    str = sprintf("%d / %d = %.2f", 22, 7, 22 / 7)
    print str
}
```

参　照

printf

算術関数

sqrt

書 式

sqrt(*x*)

機 能

*x*の平方根を返す。*x*の値は0以上でなければならない。もし0より小さい値を与えると、実行時エラーとなる。

サンプルプログラム

```
BEGIN {
    # 最後にエラーまでお見せしましょう。
    for (i = 10; i >= -1; i--) {
        printf "sqrt(%2d) = %.8f¥n", i, sqrt(i)
    }
}
```

参 照

exp, log

算術関数

srand

書　式

srand([x])

機　能

乱数の種[*1]を x に設定し、それまでの種を返す。x を省略した場合、現在の時刻が使われる。

サンプルプログラム

```
BEGIN {
    srand()
    time = srand()
    sec  = time % 60
    min  = int(time / 60) % 60
    hour = int(time / 3600 + 9) % 24
    printf "%02d:%02d:%02d¥n", hour, min, sec
}
# トリッキーになっちゃったんで、randのサンプルも見てネ！
```

注　意

乱数の種は awk が実行されるたびに、毎回ある一定の値に初期化されてしまう。これは awk を実行するたびに rand() が同じ値 (乱数列) を返すことを意味する。したがって毎回違う乱数列を得たい場合、srand() を実行して種を新しくしてやる必要がある。

参　照

rand

---

*1　計算機で乱数を得る場合、疑似的に乱数を発生させる。この時、計算で使う初期値のことを一般に種 (seed) と呼ぶ。したがって種が同じならば、同じ乱数が発生することになる。

その他

strftime

書 式

strftime(〈書式〉 [,〈時刻〉])

機 能

〈時刻〉を〈書式〉にしたがって変換した文字列を返す。〈時刻〉は1970年1月1日の0時0分(UTC)からの経過秒数で指定する。〈時刻〉が省略された場合、現在の時刻が使われる。〈書式〉で指定できるものは次のとおりである。

書式文字	意味
%Y	西暦を表す数字 (4桁)
%y	西暦の下位2桁 (00〜99)
%C	西暦の上位2桁 (00〜99)
%B	月の名称 (January, February, ...)
%b	月の省略名 (Jan, Feb, ...)
%m	月を表す数字 (01〜12)
%U	週を表す数字。その年の最初の日曜日を第1週とする (0〜53)
%W	週を表す数字。その年の最初の月曜日を第1週とする (0〜53)
%V	週を表す数字。その年の最初の月曜日を第1週とし、それ以前の曜日は前の年の最終週 (第53週) に組み込まれる (1〜53)
%A	曜日の名称 (Sunday, Monday, ...)
%a	曜日の省略名 (Sun, Mon, ...)
%u	曜日を表す数字。月曜日を1とする (1〜7)
%w	曜日を表す数字。日曜日を0とする (0〜6)
%d	日。1桁の場合は先頭に0をつける (01〜31)
%e	日。1桁の場合は先頭にスペースをつける (1〜31)
%j	年中の通算日 (001〜366)

書式文字	意味
%H	時。24 時間形式 (00〜23)
%I	時。12 時間形式 (01〜12)
%k	時。24 時間形式、1 桁の場合は先頭にスペースをつける (0〜23)
%l	時。12 時間形式、1 桁の場合は先頭にスペースをつける (1〜12)
%p	午前または午後 (AM, PM)
%M	分 (00〜59)
%S	秒。閏秒をサポートしている場合は 61 まである (00〜59)
%v	日付。VMS 形式 (例:20–AUG–1993)
%Z	タイムゾーン名 (JST など)
%n	改行
%t	水平タブ
%%	"%" そのもの

サンプルプログラム

```
BEGIN {
    print strftime("%a %b %e %H:%M:%S %Z %Y")
}
```

注　意

この関数は、gawk の 2.13 からサポートされている。

参　照

systime

文字列関数

sub

書　式

sub(〈正規表現〉,〈置換文字列〉[,〈対象文字列〉])

機　能

〈対象文字列〉中で〈正規表現〉に最初に一致する部分文字列を〈置換文字列〉に置換し、置換された数、つまり1または0を返す。〈対象文字列〉を省略した場合、代わりに$0が使われる。〈置換文字列〉中の'&'は、〈正規表現〉に一致した部分文字列に置き換えられる。〈置換文字列〉に'&'そのものを含めたい場合、'¥¥&'と書かなければならない。

gsubは〈正規表現〉に一致した部分文字列すべてを置換するのに対し、subは最初に一致した部分文字列のみを置換する。

サンプルプログラム

```
{
    sub("環境", "宗教")
    print
}
```

注　意

gsubおよびsubは他の関数と異なり、引数(〈対象文字列〉)に変更を加える。したがって〈対象文字列〉は変数でなくてはならない。

参　照

gsub

文字列関数

substr

書　式

substr(〈対象文字列〉,〈開始位置〉[,〈バイト数〉])

機　能

〈対象文字列〉中のバイト単位で指定した〈開始位置〉から〈バイト数〉分の部分文字列を返す。〈バイト数〉を省略した場合、〈対象文字列〉の〈開始位置〉から最後までを返す。

サンプルプログラム

```
{
    print detab($0, 8)
}
function detab(s, n, t, i, j) {
    for (i = 1; i <= length(s); i++)
        if (substr(s, i, 1) == "¥t") {
            while (j < n) {
                t = t " "
                j = j + 1
            }
            j = 0
        } else {
            t = t substr(s, i, 1)
            j = (j + 1) % n
        }
    return t
}
```

注　意

漢字の文字列を〈対象文字列〉として指定した場合、漢字コードを分断してしまう危険性があるので、注意すること。(jsubstr() のサンプルも参照)

参　照

jsubstr, index, length

その他

system

書 式

system(〈文字列〉)

機 能

〈文字列〉をD?OSのコマンドとして実行し、その終了コードを返す。

サンプルプログラム

```
BEGIN {
    printf "Hello %s, welcome to AWK256.¥n", ENVIRON["USER"]
    printf "AWK256% "
}

{
    if ($0 == "exit" || $0 == "quit" || $0 == "logout") {
        exit
    } else if ($0 != "" && system($0) != 0) {
        printf "¥a"
    }
    printf "AWK256% "
}

END {
    printf "Goodbye %s, see you again!¥n", ENVIRON["USER"]
}
```

その他

systime

書　式

systime()

機　能

現在の時刻を 1970 年 1 月 1 日の 0 時 0 分 (UTC) からの経過秒数で返す。

サンプルプログラム

```
BEGIN {
    a = "123456789"
    printf "print a      : " >"/dev/stderr"
    start = systime()
    for (i = 0; i < 30000; i++) print a
    end = systime()
    printf "%d sec¥n", end - start >"/dev/stderr"
}
```

注　意

この関数は、gawk の 2.13 からサポートされている。

参　照

strftime

文字列関数

tolower

書　式

tolower(〈文字列〉)

機　能

〈文字列〉中の大文字を小文字にすべて変換した文字列を返す。

サンプルプログラム

```
{
    print tolower($0)
}
```

参　照

toupper

文字列関数

toupper

書　式

toupper(〈文字列〉)

機　能

〈文字列〉中の小文字を大文字にすべて変換した文字列を返す。

サンプルプログラム

```
{
    print toupper($0)
}
```

参　照

tolower

組み込み変数

ARGC

機　能

コマンドライン引数の数。awk 自身に対するコマンドラインオプションは数に含まれない。

サンプルプログラム

```
BEGIN {
    for (i = 0; i < ARGC; i++) {
        printf "%2d: '%s'¥n", i, ARGV[i]
        if (ARGV[i] ~ /^-/) {
            delete ARGV[i]
        }
    }
}
```

参　照

ARGV, ARGIND

組み込み変数

ARGIND

機　能

現在処理中のARGVへのインデックス。ARGV[ARGIND]は現在処理中のファイル名を表す。つまりFILENAMEとARGV[ARGIND]は常に同じになる。

サンプルプログラム

```
BEGIN {
    while (ARGV[++ARGIND] ~ /^-/) {
        printf "my option: '%s'¥n", ARGV[ARGIND]
    }
}
```

注　意

この組み込み変数は、gawk 2.15からサポートされている。

参　照

ARGC, ARGV, FILENAME

組み込み変数

ARGV

機　能

コマンドライン引数の配列。配列の要素は0から始まりARGC−1までである。0番目の要素にはawkプログラム自身の名前が格納される。ARGVの内容を変更することにより、処理するファイルを変更できる。

サンプルプログラム

(ARGCを参照)

参　照

ARGC, ARGIND

組み込み変数

CONVFMT

機　能

数値を文字列に変換するときに使用される書式。デフォルト値は“%.6g”である。

サンプルプログラム

```
BEGIN {
    OFMT    = "%.6f"
    CONVFMT = "%.6e"
    pi = 22 / 7
    print pi
    print pi ""
}
```

注　意

gawk 2.12 以前では、数値から文字列への変換を、すべて OFMT によって行っていた。gawk 2.13 以降では、OFMT は print のときのみ有効で、それ以外の場所で数値の文字列への変換が行われると CONVFMT が使用されるように変更された。

参　照

OFMT, printf

組み込み変数

ENVIRON

機　能

環境変数の連想配列。この配列の内容を変更しても他のプロセスには影響がない。

サンプルプログラム

```
BEGIN {
    printf "user: %s¥n", ENVIRON["USER"]
    printf "home: %s¥n", ENVIRON["HOME"]
}
```

組み込み変数

ERRNO

機　能

getline や close でエラーが発生したとき、エラーの内容を表す文字列が格納される。

サンプルプログラム

```
BEGIN {
    fname = "/usr/games/nethack/record~"
    if ((getline < fname) == -1) {
        printf "%s: '%s'¥n", ERRNO, fname >"/dev/stderr"
        exit 1
    }
}
```

注　意

この組み込み変数は、gawk 2.15 からサポートされている。

組み込み変数

FIELDWIDTHS

機 能

この組み込み変数に値が代入されると、awkは入力レコードを固定長フィールドからなるものとして扱う。値は各フィールドの長さ（バイト単位）を空白で区切って指定する。また、指定されたフィールドの合計を越えたものについては無視される。

サンプルプログラム

```
BEGIN {
  FIELDWIDTHS = "9 4 10"
  while ("dir" |getline) {
    sub(/ *$/, "", $1)
    sub(/ *$/, "", $2)
    if ($3 + 0 != 0)
      if ($2 ~ /^$/)
        printf "%s %s¥n", $3, $1
      else
        printf "%s %s.%s¥n", $3, $1, $2
  }
}
```

注 意

この組み込み変数は、gawk 2.13からサポートされている。

組み込み変数

FILENAME

機　能

現在処理中のファイル名。標準入力から読み込んで処理している場合、FILENAMEの値は'-'になる。

サンプルプログラム

```
FNR == 1 {
    printf "¥"%s¥"¥n", FILENAME
}

{
    printf "%3d: %s¥n", FNR, $0
}
```

参　照

ARGIND

組み込み変数

FNR

機　能

現在処理中のファイル先頭からのレコード番号。

サンプルプログラム

(FILENAMEを参照)

参　照

NR, FILENAME

注　意

FNRは現在処理中のファイル中のレコード番号なのに対し、NRは全ての処理ファイルを通したレコード番号である。

組み込み変数

FS

機 能

入力フィールドセパレータの値。デフォルト値はブランク(" ")である。

サンプルプログラム

```
BEGIN {
    FS = ",[ ¥t]*"
}
```

注 意

FSの値が","のように1文字場合、文字そのものがセパレータとして扱われ、",+"のように2文字以上の場合、正規表現として扱われる。またFSの値がブランク" "の場合、特別に空白文字の繰り返し(/[¥t]+/)と解釈される。

参 照

RS, OFS, getline, split

組み込み変数

IGNORECASE

機 能

大文字小文字識別フラグ。IGNORECASEの値が0以外の場合、awkはパターンマッチの時に大文字と小文字の識別を行わず同じ文字として扱う。

サンプルプログラム

```
BEGIN {
    IGNORECASE = 1
}

/aWk/
```

組み込み変数

NF

機　能

処理中のレコードにおけるフィールドの数。

サンプルプログラム

```
{
    word += NF
}

END {
    printf "total: %d lines, %d words.¥n", NR, word
}
```

参　照

FS, getline

組み込み変数

NR

機　能

今まで処理したレコードの総数。

サンプルプログラム

(NFを参照)

参　照

FNR, getline

組み込み変数

OFMT

機　能

数値を文字列に変換するときに使用される書式。デフォルト値は“%.6g”である。

サンプルプログラム

```
BEGIN {
    print 22 / 7
    OFMT = "%.5e"
    print 22 / 7
}
```

参　照

CONVFMT, printf

組み込み変数

OFS

機　能

出力フィールドセパレータ。デフォルト値はブランク“ ”である。

サンプルプログラム

```
BEGIN {
    OFS = ","
    ORS = ";¥n"
    print "shimura", "taku", "03-3xxx-xxxx"
    print "kabe", "shozo", "0427-xx-xxxx"
    print "aoki", "tsuneo", "045-4xx-xxxx"
}
```

参　照

ORS, FS, print

組み込み変数

ORS

機　能

出力レコードセパレータ。デフォルト値は改行文字 ("¥n") である。

サンプルプログラム

(OFS を参照)

参　照

OFS, RS, print

組み込み変数

RS

機　能

入力レコードの区切り*2。RS の先頭の 1 文字だけが用いられる。デフォルト値は改行 ("¥n") である。

サンプルプログラム

```
BEGIN {
    FS = "[,;]"
    RS = "¥n"
}
{
    printf "family name:  %s¥n", $1
    printf "first name :  %s¥n", $2
}
```

参　照

FS, ORS, getline

---

*2 特別に RS の値が空文字列の場合、レコードは空行 (改行のみの行) で区切られ、入力フィールドの区切りには FS の値に加え改行が使われる。

組み込み変数

RLENGTH

機 能

直前の match 関数で指定した正規表現に一致した部分文字列の長さが格納される。一致するものがない場合、−1 が格納される。

サンプルプログラム

```
BEGIN {
  match("awk256appendix", /[0--9]+/)
  printf "RLENGTH : %d¥n", RLENGTH
  printf "RSTART : %d¥n", RSTART
}
```

参 照

RSTART, match

組み込み変数

RSTART

機 能

直前の match 関数で指定した正規表現に一致した部分文字列の位置が格納される。一致するものがない場合、0 が格納される。

サンプルプログラム

(RLENGTH を参照)

参 照

RLENGTH, match

組み込み変数

SUBSEP

機　能

配列の添字の区切り文字。デフォルト値は“¥034”である。

サンプルプログラム

```
BEGIN {
    SUBSEP = "@"
    a[1,1] = "hello"
    a[1,2] = "world"
    print a["1@1"]
    print a["1@2"]
}
```

著者紹介

志村　拓(しむら　たく)

1963年、兵庫県生まれ。本業はUNIX系のプログラマだが、その傍ら月刊アスキー誌などで執筆活動を行っている。主な著作に「VZを256倍使うための本」、「QuickBASIC4.5プログラミング入門」、「MIFES Ver.5を256倍使うための本」(いずれもアスキー出版局)などがある。最近購入したDOS/Vマシンにはまっているらしく編集部では原稿の遅れを警戒中。しかし散財のため書かないわけにはいかない？とも言われている。

鷲北　賢(わしきた　けん)

1968年、名古屋生まれ。東京工科大学を卒業し現在某大手メーカーに就職。当初、月刊アスキーでバイトしようと考えていたが下宿が遠すぎて断念。やむを得ず筆者となる。著作に「Turbo Cプログラミング入門」「MIFES Ver.5を256倍使うための本」(いずれもアスキー出版局)などがある。ある人物のせいで本書の進行が遅れている間に就職してしまい、その後姿を見せなかったため編集部の評判はあまり良くない。趣味はカレーを食べること。都内のインド料理店に足しげく通っている。

西村　克信(にしむら　かつのぶ)

1967年、京都府生まれ。鷲北と同学年のはずだが、いまだ東京工科大学在学中。「MS-DOS5アセンブラ・プログラミング」(アスキー出版局)などにプログラマとして協力。著者としては本書が初登場となる。ある人物のせいで本書の進行が遅れ、就職活動が邪魔されて結構本気であせっているらしい。KawasakiのZZ-R400でR246を疾走しアスキーへ駆けつけるが、編集部ではカレーが喰いたいだけだと言われている。

●本書の内容に関するご質問は、小社第一書籍編集部まで、封書（返信用切手同封のこと）にてお願い致します。
電話によるお問い合わせには、応じられません。
なお、本書の範囲を越える質問に関しては、お答えできない場合もあります。
●落丁・乱丁本は、送料当社負担にてお取り替え致します。
お手数ですが、小社営業部までご返送ください。

■Special Thanks：田中 良知(serow)
佐々木 毅史
■Special No Thanks：可部 正三
■編集協力：瀧井 栄子
中根 聡子
本多 清廣

AWKを256倍使うための本

1993年8月21日　初版発行
1994年10月11日　第1版第3刷発行

著　者　志村（シムラ） 拓（タク）、鷲北（ワシキタ） 賢（ケン）、西村（ニシムラ） 克信（カツノブ）
発行人　宮崎 秀規
編集人　佐藤 英一
発行所　**株式会社アスキー**
〒151-24　東京都渋谷区代々木4-33-10
振　替　00140-7-161144
大代表　(03)5351-8111
出版営業部 (03)5351-8194（ダイヤルイン）
第一書籍編集部 (03)5351-8106（ダイヤルイン）

©1993 Taku Shimura, Ken Washikita, Katsunobu Nishimura

本書は著作権法上の保護を受けています。本書の一部あるいは全部について（ソフトウェア及びプログラムを含む）、株式会社アスキーから文書による許諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。

製　作　株式会社 GARO
印　刷　共同印刷 株式会社

編　集　岡部　和成／富田　万紀子

ISBN4-7561-0162-3　　**Printed in Japan**

●11752

「AWKを256倍使うための本」別売フロッピーディスクのお知らせ

本書の4章「awk実践プログラミング」で紹介されたスクリプトを収めたディスクを販売します。

■フロッピーディスクに含まれるもの

- awk住所録（テキストデータ用DBMS）
- ここだよーんゲーム（正規表現を学ぶスクリプト）
- 姓名バリバリチューン（姓名判断逆用スクリプト）
- ユーザー管理awk（UNIXライクなログイン管理）
- ジャパンダイレクトだjpn2awk!（日本語からawkスクリプトを生成）
- 立体視awk（Random Characterでステレオグラムを）
- 表プロセッサ（アスキーコード、JIS罫線などで表を生成）

各スクリプトは、それぞれ7つの独立したディレクトリに格納され、awkのスクリプトとサンプルデータ、そしてreadme.docが収録されています。

＊注意　jgawkの本体自体は含まれませんのでご注意ください。

■ディスク収録されたデータの著作権は作者、ならびにアスキーにあります。無許可で各パソコン通信ネットにアップロードしたり、その他の方法で配付することは認められませんのでご注意ください。

申し込み方法

◇郵便振替

お近くの郵便局の窓口で郵便振替の用紙をもらってください。
用紙の表面に
①口座番号「東京9-139724」
②加入者名「株式会社　アスキー直販部」
③あなたの住所と氏名、連絡先電話番号、
裏面に、「AWKを256倍使うための本　ディスク希望」ならびにご希望のメディア3.5インチ(2DD:720K) 5インチ(2HC:1.2M)のどちらかを必ずご記入のうえ料金1500円を郵便局からお振替ください。1〜3週間程度でお手元にお届けします。

●ディスク申し込み方法についてのお問い合わせ先

株式会社　アスキー　直販部　　TEL: (03)5351-8703

●料金

1枚につき 1500円（郵送料含む/消費税込み）

●構成

3.5インチ2DDまたは5インチ2HCディスク1枚

＊ASCII NETのIDをお持ちの方は、SoftShopでこのデータを購入（ダウンロード）することができます。SoftShopに関しては、アスキーネット事務局 (TEL:03-3486-9661)へお問い合わせください。

郵便はがき

料金受取人払

代々木局承認

824

差出有効期間
平成8年4月
19日まで
郵便切手は
いりません

15124

東京都渋谷区代々木4-33-10
株式会社アスキー
第一書籍編集部 行

フリガナ
お名前　　男・女　　年齢　　歳

〒
ご住所

TEL　　FAX

勤務先名
（学校名）

お勤め先の業種

1. 製造業　2. 金融・保険・不動産業　3. 商社・卸・小売業　4. 建設業
5. 運輸・通信・公共サービス業　6. その他サービス業　7. 政府・公共機関
8. 学校・研究所・病院　9. 農林水産業　10. 情報産業　11. 学生
12. その他

職種

1. 経営・社業全般　2. 総務・庶務　3. 経理・財務　4. 人事・労務
5. 企画・調査通信　6. 営業・販売　7. 広報・宣伝　8. 設計・デザイン
9. 資材・購買　10. 生産・工事　11. 研究・開発・技術　12. 情報処理・EDP
13. 教育　14. その他

1. 本書を何からお知りになりましたか？

☐ 新聞の広告　☐ アスキーの雑誌広告　☐ アスキー以外の雑誌広告
☐ アスキーの図書目録　☐ 知人の紹介　☐ パソコンショップ
☐ 書店　☐ チラシ　☐ その他（　　　　　　　　　　）

2. 本書の評価

内容　☐満足　☐普通　☐不満　読みやすさ　☐満足　☐普通　☐不満
価格　☐満足　☐普通　☐不満　表紙　☐満足　☐普通　☐不満

3. 本書の購入先

☐書店　☐パソコンショップ　☐生協　☐その他（　　　　　　　　）

4. お使いのコンピュータをお書きください

デスクトップ　☐PC-9801シリーズ　（　　　　　　）
☐PC-9821シリーズ　（　　　　　　）
☐AT互換機（DOS/Vマシン）　（　　　　　　）
☐Macintosh　（　　　　　　）
ブック/ノート　メーカー名（　　　　　　）
その他　（　　　　　　　　　　）

5. お使いの周辺機器がありましたらお書きください

☐HD（　　MB）　☐CD-ROM　☐MOドライブ　☐モデム
☐プリンタ　☐その他（　　　　　　　　）

6. ご意見、ご感想がありましたらお聞かせください

ご協力ありがとうございました。

3.5インチ2DDまたは5インチ2HCディスク1枚

* ASCII NETのIDをお持ちの方は、SoftShopでこのデータを購入（ダウンロード）することができます。SoftShopに関しては、アスキーネット事務局（TEL:03-3486-9661）へお問い合わせください。

9784756101624

1913055012005

ISBN4-7561-0162-3

C3055 P1200E

ASCII
256倍

定価1,200円（本体1,165円）